BEIJING NORMAL-HONG KONG BAPTIST UNIVERSITY

Doctor of Philosophy

THESIS ACCEPTANCE

DATE: March 25, 2025

STUDENT'S NAME: HUANG Junhao

THESIS TITLE: On Modular Arithmetic and Polynomial Multiplication in Lattice-based Cryptography

This is to certify that the above student's thesis has been examined by the following panel members and has received full approval for acceptance in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Chairman:	Prof Su Weifeng Professor, Faculty of Science and Technology, BNBU (Designated by Dean of Faculty of Science and Technology)
Internal Members:	Prof Xuanyuan Zhe Professor, Faculty of Science and Technology, BNBU (Designated by Head of Department of Computer Science)
	Dr Fan Wentao Associate Professor, Faculty of Science and Technology, BNBU
External Examiners:	Prof He Debiao Professor School of Cyber Science and Engineering Wuhan University
	Prof Yang Anjia Professor College of Cyber Security Jinan University
In-attendance:	Dr Chen Donglong Donald Associate Professor, Faculty of Science and Technology, BNBU
	Dr Zhu Shuyan Associate Professor, School of Microelectronics Science and Technology Sun Yat-sen University

Issued by Graduate School, BNBU

On Modular Arithmetic and Polynomial Multiplication in Lattice-based Cryptography

HUANG Junhao

A thesis submitted in partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

Principal Supervisor:

Dr. Donglong CHEN

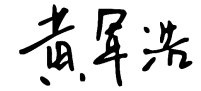
(Beijing Normal-Hong Kong Baptist University)

March 2025

DECLARATION

I hereby declare that this thesis represents my own work which has been done after registration for the degree of PhD at Beijing Normal-Hong Kong Baptist University, and has not been previously included in a thesis or dissertation submitted to this or any other institution for a degree, diploma or other qualifications.

I have read the University's current research ethics guidelines, and accept responsibility for the conduct of the procedures in accordance with the University's Research Ethics Committee (REC). I have attempted to identify all the risks related to this research that may arise in conducting this research, obtained the relevant ethical and/or safety approval (where applicable), and acknowledged my obligations and the rights of the participants.

Signature: 

March 2025

ABSTRACT

Post-quantum Cryptography (PQC) is designed to be a quantum-safe variant of the Public Key Cryptographic (PKC) schemes, as the advancement of quantum computing poses a significant threat to the traditional PKC. Among all the NIST PQC candidates, Lattice-based Cryptography (LBC) offers both strong security and competitive performance, making it a leading contender in the NIST PQC standardization project. With the rapid expansion of the Internet of Things (IoT) across various industries, securing vast amounts of sensitive data on IoT devices using the post-quantum cryptography has become increasingly critical.

However, IoT devices are often constrained by limited power, memory, and computational capacity. Deploying NIST PQC standards, particularly the most promising LBC schemes, on these resource-restricted devices presents significant challenges. To address these challenges, this thesis aims to develop efficient, lightweight, and secure implementations of lattice-based cryptography for IoT devices. This research focuses on both algorithmic design and practical implementations.

From an algorithmic design perspective, this thesis proposes an improved Plantard arithmetic for signed integers and further enlarges the input range of the Plantard arithmetic without modifying its computation steps. Theoretical proofs of correctness for the improved Plantard arithmetic are provided from two perspectives, demonstrating its validity. The advantages of the improved Plantard arithmetic over its original version, as well as over Montgomery and Barrett arithmetic, are thoroughly discussed. These advantages make it feasible to replace the state-of-the-art Montgomery and Barrett arithmetic with the improved Plantard arithmetic in LBC

schemes on certain platforms.

Based on the theoretical improvement of the improved Plantard arithmetic, this thesis proposed efficient implementations of the improved Plantard arithmetic on three IoT platforms with different instruction set architectures (ISA): Cortex-M4, Cortex-M3, and RISC-V. Furthermore, we show that the proposed Plantard arithmetic can be easily adapted to other schemes or platforms with similar ISA characteristics, highlighting its extensibility. By replacing the state-of-the-art Montgomery and Barrett arithmetic with the proposed Plantard arithmetic, we obtain around 20% speedup for the NTT and INTT computations, which leads to faster Kyber, NTTRU, and Dilithium implementations on the three IoT platforms. Additionally, we propose two optimization techniques on ARMv7-M, namely lazy rotations and memory access pipelining, to enhance the performance of Keccak. These optimizations bring over 20% performance improvement for the Keccak permutation on ARMv7-M.

We then investigate the optimized implementation of the masking-friendly LBC scheme, Raccoon, on the Cortex-M4, ensuring robust quantum-safe protection and side-channel resistance on memory-constrained devices. We introduce a 5-instruction negative double Montgomery reduction approach for the multi-moduli NTT implementation of Raccoon, where each 64-bit coefficient requires double modular reductions, Additionally, we demonstrate that the lazy reduction strategy is not only applicable to the NTT/INTT implementation but also enhances the efficiency of four critical masking subroutines in Raccoon. Lastly, we also explore memory optimization techniques for the lightweight implementation of the speed-optimized Kyber and high-order Raccoon, addressing the challenges imposed by memory-constrained IoT devices, thereby improving the feasibility of deploying these quantum-safe algorithms in real-world IoT environments.

Keywords: Cryptographic Engineering, Post-quantum Cryptography, Lattice-based Cryptography, Polynomial Multiplication, Modular Arithmetic

ACKNOWLEDGMENTS

Finally! Before my thirtieth birthday arrives, I have achieved the highest academic degree one can obtain in a lifetime. And I can finally change my title from Mr. Huang to Dr. Huang. As I write these words, I am overwhelmed with emotions, and tears stream down my face uncontrollably—tears of joy, tears of hardship, tears that mark the perfect conclusion to my 23-year study journey.

Looking back on these past two decades, I never imagined that I would one day pursue a PhD. Growing up in a small village, all I knew was that if I didn't study hard, my only future might be working in a factory, tightening screws. I've done that kind of work before, so I understood deeply that I had to break free from that fate. For 23 years, I studied relentlessly, with only one goal—to bid farewell to my past self. Everything I have today is the best reward for my perseverance. Now, I can stand tall and say to my past self: Thank you for never giving up. You've done incredibly well.

I know that PhDs are common nowadays, and earning one is no longer something particularly extraordinary. Some may even say I've been so single-minded in my studies that I've become foolish. But I believe that dedicating oneself to something for so long is an achievement in itself. There's no need to care about others' opinions—what truly matters is how you feel about yourself. The road ahead is still long. Keep doing what you love, and one day in the future, just like today, you will look back and be grateful for the perseverance that brought you here.

During my four-year academic research journey, I received immense support from many excellent researchers, and I would like to express my heartfelt gratitude

to them here.

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Donglong CHEN. He is not only a dedicated advisor but also a meticulous researcher and a compassionate mentor. Without his unwavering support and invaluable guidance throughout my four-year PhD journey, I would not have been able to accomplish what I have today.

At the very beginning of my research journey, he provided us with systematic guidance on how to conduct research and stay updated with cutting-edge advancements. He frequently shared state-of-the-art papers related to our research topics, inspiring us to pursue high-quality research. I still vividly remember the day he introduced us to the paper "Efficient Word Size Modular Arithmetic", presented by Thomas Plantard in 2021. It immediately struck me as a seminal work that Plantard arithmetic could be the state-of-the-art modular arithmetic in current LBC implementations. Through continuous collaboration and refinement with my supervisor and fellow lab members, (Haosong Zhao and Jipeng Zhang), we tackled numerous theoretical and implementation challenges, which ultimately led to our publication in TCHES 2022. Without Dr. Chen's consistent encouragement, insightful guidance, and strong support, I might have still been wandering aimlessly in research. As I progressed in my academic journey, he further supported me in joining Prof. Ray C. C. Cheung's CALAS research lab at City University of Hong Kong for academic exchange. This invaluable experience allowed me to connect with brilliant minds and foster meaningful collaborations. For all his mentorship, support, and kindness, I extend my heartfelt appreciation to Dr. Chen. I would also like to extend my special thanks to Prof. Ray for his tremendous support during my six-month exchange at City University of Hong Kong. His guidance provided me with the invaluable opportunity to experience the international academic environment of CityU.

Secondly, I would like to thank all my co-authors for their support and assistance in my research. Special thanks to Jipeng for his help in configuring the IoT development environment, which enabled the successful execution of the experimental

part of the TCHES 2022 paper. I am also grateful to Haosong for his contributions to the mathematical derivations of Plantard arithmetic, which helped refine part of the theoretical proof of the improved Plantard arithmetic. I would also like to thank Alexandre for accepting my collaboration invitation, which led to the successful publication of the TCHES 2024 paper. I also appreciate Zewen for the engaging academic discussions at CityU and for adapting Plantard arithmetic to his custom RISC-V platform.

Lastly, I convey my deepest gratitude to all my co-authors for their valuable suggestions and strong support in my research, especially Prof. Koç, Prof. Liu, Prof. Zhou, Dr. Dai, and Dr. Liu. Last but not least, many thanks to the committee, including Prof. Su, Prof. He, Prof. Xuanyuan, Prof. Fan, Prof. Zhu, and Prof. Yang, for their valuable suggestions to further improve the quality of the thesis.

Besides academic research, many family members and friends have given me immense encouragement, love, and companionship throughout my PhD journey. Their support has helped me navigate these four long years and overcome the ups and downs along the way. Here, I would also like to express my sincerest gratitude to them.

First and foremost, I would like to thank my mother and my siblings for their unwavering support. It is their encouragement that has allowed me to pursue the path I truly desire, enabling me to move forward without hesitation. Their support has freed me from worries, allowing me to fully dedicate myself to my academic journey. It is precisely because of them that I have been able to achieve what I have today. I dedicate this dissertation to them.

Secondly, I would like to express my heartfelt gratitude to my dearest friends who have always been by my side. Your companionship and support have shaped me into who I am today. I am grateful to Yanggui and Ruiqi for bringing joy to my PhD journey by taking me out to have fun, allowing me to experience the pleasures of life beyond research. My sincere thanks to Yangyi for accompanying me on the trips to Lingting Island, Shanghai and Zhoushan, where I was able to immerse myself in

the breathtaking beauty of nature. I truly appreciate Sinian for exploring countless delicious foods with me in Zhuhai over the past year. A special thank you to Xinshu for attending my PhD defense in person, offering me tremendous encouragement. I am also grateful to Rongji for the wonderful times we spent traveling around Zhongshan and Zhuhai. Finally, my heartfelt thanks to Yongzhen, Lirang, Zhiwei, Zhiyue, and all my dear friends for sharing with me their experiences and joys along this journey of growth. These shared moments have filled my life with happiness and gratitude.

Lastly, I would like to express my gratitude to all the outstanding fellow students and friends I met during my PhD journey. Special thanks to the "Maker Center Team": Haosong, Shiqi, and Jie for being my meal companions during the first two years of my PhD. I truly cherished those days filled with casual chats and gossip, which helped me get through a challenging yet fulfilling and joyful period. I am also grateful to my friends Danlei, Wenhao, and Songze, who always took me out to have fun in Zhuhai, making my PhD journey much more enjoyable. Many thanks to Zewen, Songyang, Gaoyu, Guangyan, and all the CALAS members at CityU for their support and engaging academic discussions. I also appreciate Zuchao, Meng Chen, Hao Miu, and Yuhang for showing me around Shenzhen and Hong Kong during my exchange period. Finally, I want to thank Sinian and Yuxuan for being my new meal companions during the last two years of my PhD, through which we built a strong bond as fellow labmates.

Once again, I would like to thank myself for never giving up. Remember to keep pursuing what you love. I hope you will find the right path for yourself! Never forget the hardships of the journey. No matter what you do, always persevere!

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

List of Acronyms

PKC	Public Key Cryptography
PQC	Post-Quantum Cryptography
LBC	Lattice-based Cryptography
NIST	the US National Institute of Standards and Technology
IoT	Internet of Things
(I)NTT	(Inverse) Number Theoretic Transform
ISA	Instruction Set Architecture
SVP	Shortest Vector Problem
CVP	Closest Vector Problem
SIS	Short Integer Solution
LWE	Learning With Errors
CRT	Chinese Remainder Theorem
RSIS	Ring Short Integer Solution
RLWE	Ring Learning With Errors
MSIS	Module Short Integer Solution
MLWE	Module Learning With Errors

CPA	Chosen Plaintext Attacks
CCA	Chosen Ciphertext Attacks
PKE	Public Key Encryption
KEM	Key Encapsulation Mechanism
DSA	Digital Signature Algorithm
(Q)ROM	Quantum Random Oracle Model
CT	Cooley-Tukey Butterfly
GS	Gentleman-Sande Butterfly
FNT	Fermat Number Transform
CPU	Central Processing Unit
FP	Floating Point
SIMD	Single Instruction Multiple Data
(S)RAM	(Static) Random Access Memory
CCM	Core Coupled Memory
XKCP	eXtended Keccak Code Package
DFT	Discrete Fourier Transform

Chapter 1

Introduction

Quantum computers are being developed rapidly. Google [16] reported their 53-qubit processor in 2019, claiming that their quantum processor can achieve quantum supremacy. Two years later, a Chinese research team [112] announced a 66-qubit processor named *Zuchongzhi*. Also in 2021, IBM announced their 127-qubit quantum processor called Eagle [66], which first breaks the 100-qubit barrier. And Later in 2022, they further unveiled their 433 qubit Osprey processor [67]. In 2024, Google Quantum AI team just announced their latest 105-qubit quantum chip, Willow [7], and claimed that it could finish a benchmark computation that would take today's fastest supercomputers 10^{25} years in just five minutes.

With the emergence and rapid development of quantum computing technology, the conventional Public Key Cryptographic (PKC) schemes face a significant threat. Shor's algorithm on quantum computers can solve the mathematical hard problems of these PKC schemes, such as the big number factorization problem of RSA, the Discrete Logarithm Problem (DLP) of ElGamal, and the Elliptic Curve Discrete Logarithm Problem (ECDLP) of Elliptic Curve Cryptography (ECC), in polynomial time. Theoretically, a quantum computer with a few thousand qubits should break the 2048-bit RSA. However, there are still noise issues not being well understood. A recent research [53] shows that factoring 2048-bit integer in RSA within 8 hours needs a quantum computer with about 20 million noisy qubits. This implies the current progress of quantum computers is still far from breaking the currently deployed 2048-

bit RSA or 256-bit ECC. On the other hand, IBM's roadmap of quantum computers [59] claims that the number of qubits could be more than doubled per year, making a one million-qubit machine possible in 2030. The progress of experimental quantum computers and the surprising developments in the quantum computing motivated the cryptographic community to expedite the study of quantum-safe cryptographic algorithms to find suitable alternatives to the traditional public-key cryptosystems within a decade or less.

1.1 Post-quantum cryptography

Post-quantum cryptography refers to public-key cryptographic (PKC) algorithms designed to be secure against threats posed by quantum computers. To promote the development of post-quantum cryptography (PQC), NIST launched a standardization project in 2016 to solicit, evaluate, and standardize post-quantum cryptographic algorithms [88]. The project received 82 candidate submissions [9]. These candidate algorithms can be divided into the following five categories: code-based [105], lattice-based [31, 42], multivariate-based [41], hash-based [25], and isogeny-based [101, 47].

In 2019, NIST selected 26 candidate algorithms that met the security and performance criteria and advanced them to the second round for further evaluation. In the third round of the standardization process, seven finalists, including four key encapsulation mechanisms (KEM) and three digital signature algorithms (DSA), have been selected to advance to the fourth round of evaluation, as shown in Table 1.1. Among these algorithms, three out of the four KEM finalists (i.e., Kyber [17], Saber [22], and NTRU [34]) are from lattice-based cryptography (LBC). In 2022, NIST announced four finalists to be standardized for its six-year Post-Quantum Cryptography (PQC) standardization project [88]. Among them, Kyber [17] is the only KEM standard, while Dilithium [42], Falcon [98], and sphincs [26] are three digital signature standards, and Kyber, Dilithium, and Falcon are all based on the lattice-based construction, which indicates that LBC is competitive and con-

Table 1.1: NIST PQC standardization project: Round-3 and Round-4 candidates.

Round	**Round 3**		**Round 4**	
Types	KEM	DSA	KEM	DSA
Schemes	Kyber	Dilithium	Kyber	Dilithium
	Saber	Falcon	-	Falcon
	NTRU	Rainbow	-	SPHINCS$^+$
	Classic McEliece	-	-	-

vincing in terms of security and performance. In 2024, three of the finalists Kyber, Dilithium, and SPHINCS$^+$ have been standardized as ML-KEM, ML-DSA, and SLH-DSA in FIPS 203 [90], FIPS 204 [91], and FIPS 205 [92]. Consequently, the efficient implementation of these NIST PQC standards, ensuring a smooth transition from conventional public-key cryptography to post-quantum cryptography algorithms in practical application scenarios, has become an important area of research.

Two out of the three finalists for the Digital Signature Algorithm (DSA) in the NIST PQC standardization project, Dilithium and Falcon, are constructed over structured lattices. To diversify the DSA standards, NIST initiated an additional round for digital signature schemes in 2022 [89], aiming to identify general-purpose signature schemes that are not based on structured lattices, or that feature short signatures and fast verification, making them particularly suitable for certain applications. Nevertheless, lattice-based DSA schemes are still welcome if they significantly outperform Dilithium and Falcon, or offer substantial security properties. Most of the additional round digital signature schemes still belong to the five categories as in the NIST PQC standardization project in 2016, i.e., code-based, isogeny-based, lattice-based, multivariate-based, and symmetric-based schemes. A novel category called MPC-in-the-head signatures that is built upon the Multi-Party-Computation-in-the-Head (MPCitH) construction has also attracted many attentions. There are still 7 lattice-based DSA schemes submitted in the first round of additional signatures [89], but only one lattice-based DSA scheme Hawk [43] has entered the second

round [89] in 2024.

Evaluating the performance of the NIST PQC standards and candidates is an essential task [10] not only during the process of the NIST standardization project but also in the future commercial deployment. The theoretical security guarantees of PQC schemes are important, but the practical deployment requires algorithms to be efficient in terms of speed, memory usage, and power consumption. Regular performance evaluations and optimizing the performance of the PQC algorithms ensure that the chosen PQC standards can meet the diverse requirements of real-world applications, such as IoT devices, mobile platforms, and large-scale server infrastructures. Besides, it also helps identify which algorithms perform best for specific use cases and ensure that the final NIST standards provide the most efficient solutions for a broad range of applications. Therefore, this thesis will also focus on the performance optimization of PQC algorithms, attempting to optimize them from both algorithm design and implementation perspectives, in order to improve their performance and enhance their practicality in real-world platforms. Since LBC schemes have dominated the NIST PQC standardization project, with three out of the four NIST PQC standards being based on LBC, this thesis will primarily focus on the efficient implementation of LBC schemes. The goal is to provide insights that could facilitate the commercial deployment of these PQC standards in the future.

1.2 PQC on the Internet of Things

The application of the Internet of Things (IoT) has spread out in various fields, such as smart healthcare, home, and transportation domains. Increasingly, IoT devices have been utilized to gather sensitive data from individuals, corporations, military organizations, and governments. It is estimated that by 2030, there will be around 30 billion interconnected IoT devices [108]. With the rapid development of quantum computing, safeguarding sensitive data on numerous IoT devices against quantum computers is a pressing concern in the near future. Therefore, deploying NIST PQC standards and candidates for these IoT devices is essential.

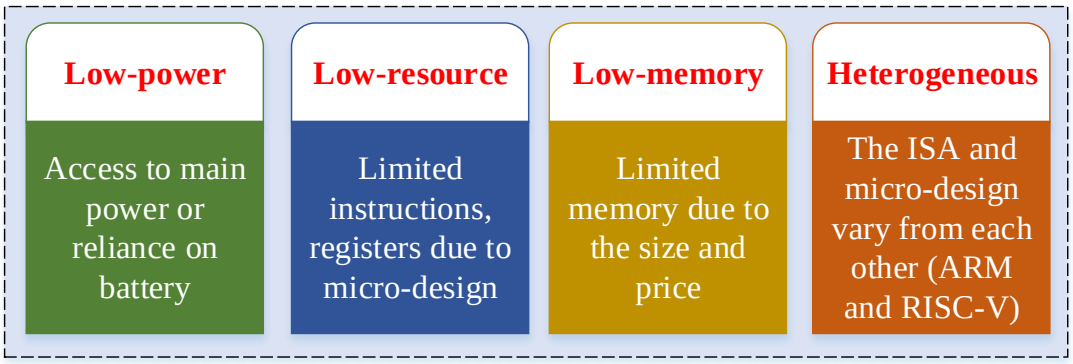


Figure 1.1: Four major limitations of IoT devices

However, the microprocessors used in IoT devices are distinct from the traditional CPUs used in computers as they are typically smaller in size and used to collect and transmit data with sensors. Therefore, they have the following limitations, as shown in Figure 1.1, which present different challenges for the PQC deployment on the IoT devices.

First of all, IoT devices are designed to be energy-efficient because they often operate in environments where power resources are limited. Many IoT devices either need constant access to a main power supply or rely on batteries for energy. To prolong battery life and reduce power consumption, the deployment of PQC algorithms on these microprocessors must be optimized for low power usage. Reducing the cycle counts of PQC algorithms is the first priority in order to reduce power consumption.

In addition, many IoT devices are constrained by limited computational resources and memory. Due to their small size and low cost, these devices are often designed with minimal instruction sets, fewer registers, and limited storage capacity, which restricts their ability to handle large data sets or perform complicated cryptographic operations required by PQC algorithms. Many of the post-quantum cryptographic (PQC) algorithms require larger key sizes and more memory (both stack and heap usage) compared to traditional public-key cryptography (PKC) algorithms like Elliptic Curve Cryptography (ECC). This makes it even more challenging to replace PKC with PQC on resource-constrained devices. As a result, deploying PQC algo-

rithms on these IoT platforms requires tailoring them to fully utilize the available architectural characteristics, such as the specific instruction sets, registers, and memory capabilities. Optimizing the PQC algorithms for these constraints is critical to achieving acceptable performance and ensuring that the cryptographic operations can run within the limits of the platform. Moreover, tailoring the PQC algorithms for limited memory may require making trade-offs between memory usage and performance. Different platforms will have different trade-offs depending on their specific resources. Therefore, careful consideration must be given to optimize the balance between memory consumption and execution time when deploying PQC algorithms on IoT devices.

Furthermore, IoT devices are heterogeneous in nature. Different devices may have different instruction set architectures (ISAs) and microarchitectures. For example, some may use ARM-based cores, others may rely on RISC-V or AVR cores. These differences impact the way in which PQC algorithms should be implemented to achieve optimal performance. A one-for-all approach is not suitable, and tailored solutions must be developed to take advantage of the specific characteristics of each platform.

The above constraints on these platforms lead to varying implementation requirements and pose unique challenges for the deployment of PQC. As both memory consumption and efficiency are critical factors when assessing the feasibility of PQC on IoT devices, the primary challenge for the PQC implementation on IoT devices is to explore an efficient and lightweight PQC implementation tailored for heterogeneous IoT platforms. This includes efficiently utilizing the constrained computational resources of the target platforms, achieving reasonable efficiency to minimize power consumption, while also ensuring that stack usage remains small enough to fit within the limited memory. Balancing these competing factors is essential for the successful and practical deployment of PQC algorithms on resource-constrained IoT platforms.

1.3 Cryptographic engineering

Cryptographic engineering is the field focused on the design, implementation, and analysis of cryptographic systems to ensure secure communication and data protection. It bridges the gap between theoretical cryptography and practical system applications, addressing challenges such as efficiency, real-world constraints, and implementation security. Since cryptographic engineering involves both software and hardware implementations, it requires expertise in electrical engineering and computer science. Furthermore, because most cryptographic algorithms, especially public-key cryptography like RSA [102], ECC [111], and PQC [9], rely on computationally intensive mathematical operations, a strong mathematical foundation is essential. Thus, cryptographic engineering can be seen as an interdisciplinary field, integrating electrical engineering, computer science, and mathematics [75].

In this thesis, the implementation of PQC schemes adheres to the following objectives of cryptographic engineering:

- **Efficiency.** Efficiency is crucial for the practical deployment of cryptographic systems in cryptographic engineering. Especially for public-key cryptography, which involves complex mathematical computations, cryptographic algorithms can be computationally intensive. Optimizing their performance without compromising security is essential for offering efficient and secure protection of the sensitive data in real-world applications. The efficiency requirements for cryptographic algorithms can vary significantly depending on the target platform. For instance, on server platforms, handling large volumes of cryptographic computations is often a priority. In such cases, high-throughput cryptographic implementations are crucial to alleviate the burden on servers and ensure smooth, large-scale processing of cryptographic tasks [50, 35]. In contrast, on client platforms or resource-constrained devices, such as mobile phones or IoT devices, minimizing the latency of each cryptographic operation becomes more important to enhance the user experience. Fast and responsive cryptographic operations ensure that users do not encounter delays while interacting with

secure applications, such as encrypted messaging or financial transactions.

- **Real-world Constraints.** The deployment of cryptographic algorithms might also be affected by real-world constraints. Different platforms may also have different constraints. For example, PQC on IoT devices presents unique challenges due to their limited resources, including power, computational capacity, and memory. These devices are often designed with energy efficiency in mind, which limits their ability to perform computationally intensive cryptographic operations, such as those required by PQC schemes. IoT devices typically operate on battery power and have limited processing resources such as registers and instruction sets. Optimizing these operations for the specific architecture of IoT devices (e.g., ARM Cortex-M, RISC-V) is crucial to achieve acceptable performance without excessive power or time overhead. Memory is another significant constraint in resource-constrained IoT devices. Many PQC schemes require large public and private keys and consume a large stack usage, making them memory-intensive. Therefore, minimizing memory usage while ensuring security and performance is key to making these cryptographic schemes practical in constrained IoT devices.

- **Implementation Security.** In cryptographic engineering, even if cryptographic algorithms are secure from a theoretical standpoint, real-world implementations can still be vulnerable to attacks if proper precautions are not taken. A major threat to secure cryptographic implementations arises from side-channel attacks. These attacks exploit physical characteristics or unintended information leakage, such as variations in timing [77], power consumption [76], and electromagnetic (EM) emissions [6], during the execution of cryptographic algorithms to extract sensitive data, such as secret keys or plaintext [107]. One of the most effective ways to mitigate timing attacks is to implement cryptographic algorithms in constant-time. This means that the time taken by the algorithm should not depend on the input data, especially the secret key [77]. For example, modular exponentiation in RSA,

scalar multiplication in ECC, and polynomial multiplication in LBC should be implemented using constant-time algorithms to prevent the timing variations. Moreover, techniques such as masking or randomization can be used to make the power consumption independent of the secret data being processed. This is done by adding random noise to the computation or by splitting the sensitive data into multiple parts and combining the results [76, 68].

Regarding the first two objectives, this thesis optimizes PQC schemes from both algorithm design and implementation perspectives, ensuring an efficient and lightweight deployment on resource-constrained IoT devices. For the third objective, this thesis ensures implementation security by adopting constant-time implementations, avoiding secret-dependent branching as discussed in Chapter 4, and exploring masking techniques in Chapter 5. These strategies guarantee that the execution time or power patterns of cryptographic operations remain independent of the secret data, thereby mitigating the risk of some side-channel attacks.

1.4 Contributions

Guided by the objectives of cryptographic engineering, the primary goal of this thesis is to develop efficient, lightweight, and secure implementations of post-quantum cryptography tailored for Internet of Things devices. This research addresses both the algorithmic design and the implementation aspects to ensure practical deployment in resource-constrained environments. This thesis specifically focuses on lattice-based cryptography, as three out of the four NIST PQC standards—Kyber, Dilithium, and Falcon —are based on the lattice-based hard problems. These schemes have garnered significant attention due to their strong security guarantees and promising performance in the face of emerging quantum threats. As such, the work in this thesis aims to enhance the efficiency of LBC algorithms for IoT devices, ensuring that they meet both the performance and security requirements of the next-generation cryptographic standards.

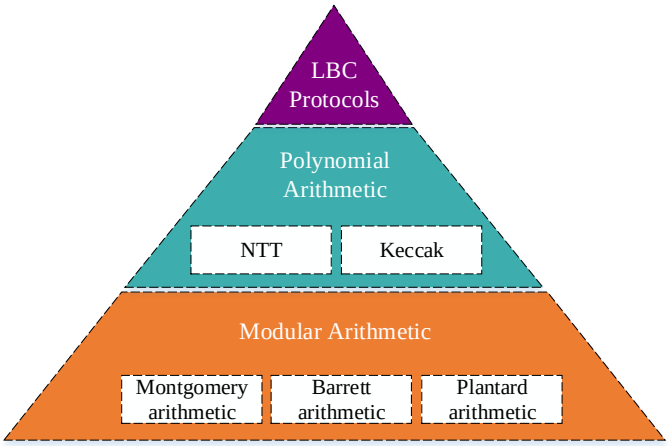


Figure 1.2: An overall structure of the LBC schemes

Overall structure of the LBC schemes. As illustrated in Figure 1.2, the LBC schemes can be broadly divided into three layers: modular arithmetic, polynomial arithmetic, and the LBC protocol. The most fundamental layer is the modular arithmetic over $\mathbb{Z}_q$, which includes operations such as modular addition, subtraction, multiplication, and reduction. Modular arithmetic is essential for NTT computations in LBC schemes. The current state-of-the-art techniques for modular arithmetic include Montgomery arithmetic, Barrett arithmetic, and specialized methods tailored to the specific form of the modulus q. The second layer, polynomial arithmetic, is crucial as most operations in LBC schemes are carried out over the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. This includes polynomial addition, subtraction, multiplication, and sampling, which contribute significantly to the running time of LBC schemes. The polynomial sampling in LBC schemes normally relies on the SHA-3 function that is constructed over Keccak permutation. Finally, the LBC protocol layer provides the cryptographic primitives necessary for practical applications, such as key generation, key encapsulation, key decapsulation, signature generation, and signature verification.

Optimization overview. Similarly, the optimizations presented in this thesis focus on three distinct layers, as illustrated in Figure 1.3.

- The first and fundamental optimization of this thesis focuses on modular arith-

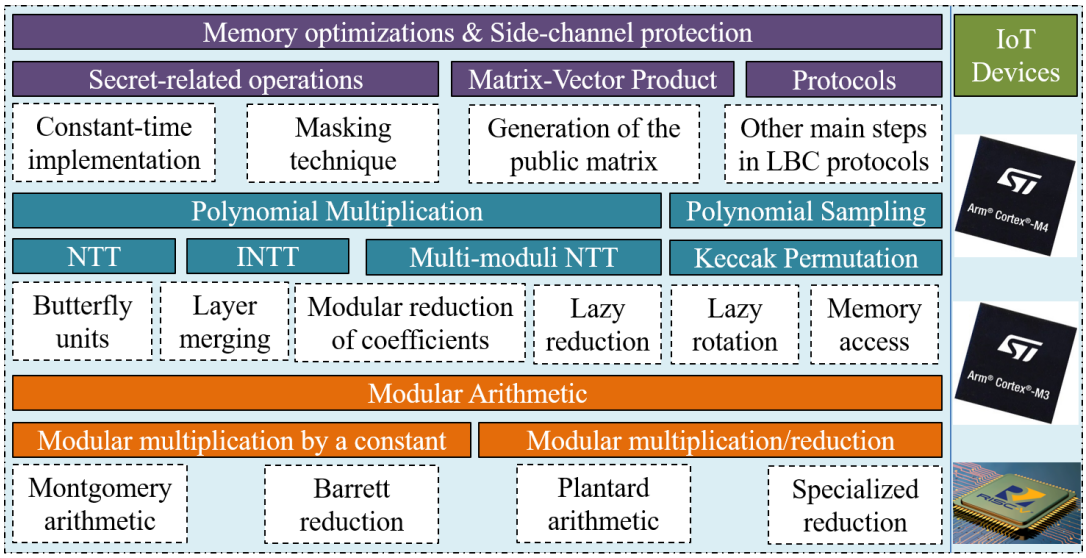


Figure 1.3: An overview of the optimization techniques in LBC schemes

metic. We analyze state-of-the-art modular arithmetic in LBC schemes across different scenarios and propose an improved Plantard arithmetic, specifically tailored for lattice-based cryptography. The proposed Plantard arithmetic demonstrates significant advantages over the state-of-the-art Montgomery and Barrett arithmetic, making it possible to supersede these older modular arithmetic in LBC on certain platforms.

- The second layer addresses the optimization of two core operations critical to LBC schemes: polynomial multiplication and the Keccak permutation. By integrating the improved Plantard arithmetic with various optimization techniques in polynomial multiplication, we highlight its superiority in NTT implementation. We also present two architectural optimizations for Keccak permutation on ARMv7-M. These refinements further improve the overall performance of LBC, making it more efficient and practical for real-world applications.

- The final layer emphasizes memory optimization and side-channel protection for LBC, a critical factor in deploying LBC schemes on IoT devices. Given the resource constraints of IoT devices and their susceptibility to side-channel attacks, it is crucial to minimize memory consumption while ensuring robust

side-channel resistance for the practical and secure deployment of PQC in such environments.

Contributions. More specifically, the detailed contributions of this thesis are summarized as follows:

- First, we present an improved Plantard arithmetic for signed integers with its correctness proof. We then further enlarge the input range of the Plantard arithmetic without modifying its computation steps, and give the proof. To provide a comprehensive understanding of Plantard arithmetic, we introduce another variant of signed Plantard arithmetic proposed in a concurrent work [15]. Additionally, we present an optimization technique to eliminate one rounding operation for the signed Plantard arithmetic in [15], enhancing its efficiency on platforms that lack inherent support for right-shifting with rounding approximation. Finally, we detail the advantages of the improved Plantard arithmetic over its original version, signed Plantard arithmetic in [15], Montgomery and Barrett arithmetic. Compared to the original Plantard arithmetic, the improved Plantard arithmetic not only supports signed integers but also extends its input range and narrows down its output range, thus enabling better lazy reduction strategies in NTT/INTT. Besides, it inherits the advantage of the original Plantard multiplication, i.e., saving one multiplication when one of the two operands is a constant. We further demonstrate that the improved modular arithmetic also has excellent merits over the state-of-the-art Montgomery and Barrett arithmetic, making it possible to supersede the Montgomery and Barrett arithmetic with the improved Plantard arithmetic in LBC schemes on certain platforms.

- Based on the theoretical improvement of the improved Plantard arithmetic, we present faster implementations of the improved Plantard arithmetic for 16-bit odd moduli on Cortex-M4, Cortex-M3 and RISC-V. Furthermore, we show that the Plantard arithmetic can be easily extended to other schemes or

platforms with similar ISA characteristics, which highlights its extensibility. We then integrate the improved Plantard arithmetic into the 16-bit NTT in Kyber and NTTRU, as well as the multi-moudli NTT in Dilithium. For the 16-bit polynomial multiplication in Kyber and NTTRU, by replacing the state-of-the-art Montgomery and Barrett arithmetic with the proposed Plantard arithmetic, we demonstrate that it enables more efficient polynomial arithmetic, namely NTT, INTT, and modular reduction, leading to faster Kyber and NTTRU implementations on the above three 32-bit platforms. As for the polynomial multiplication in Dilithium, which typically uses 32-bit NTT, we revisit the idea of using small NTT (i.e., 16-bit NTT) for $c\mathbf{s}_i$ and multi-moduli NTT for $c\mathbf{t}_i$ on ARMv7-M. We show that by leveraging the efficient Plantard arithmetic, the proposed multi-moduli NTT implementation on Cortex-M3 remains constant-time and can indeed yield over 19.07% or 52.76% speed-ups when compared with the variable-time or constant-time 32-bit NTT in [57], respectively. Moreover, we also show that the efficient Plantard arithmetic can be used to accelerate the explicit CRT by 19.45% in the multi-moduli NTT implementation, further improving the feasibility and efficiency of the multi-moduli NTT on Cortex-M3. The optimized implementations demonstrate that the Plantard arithmetic can indeed supersede the state-of-the-art Montgomery and Barrett arithmetic on these low-end 32-bit microprocessors.

- As for another core operation in LBC schemes, the polynomial sampling, we improve Keccak's performance on Cortex-M3 and M4 by up to 24.78% and 21.4% utilizing two architecture-specific optimizations. The first one, denoted as lazy rotations, consists of taking advantage of the inline barrel shifter to eliminate explicit rotations in the linear layer. Note that this technique has been reported in the literature by Becker and Kannwischer in order to boost Keccak on ARMv8 architectures [24] but has not been ported to ARMv7-M so far. The second optimization consists of a more efficient memory access scheduling to avoid pipeline hazards, which results in over 12.84% performance

improvements on ARMv7-M.

- We then present the efficient implementation of the masking-friendly Raccoon, offering robust and efficient countermeasures to secure against side-channel attacks. In Raccoon's multi-moduli NTT implementation, the double modular reductions are required for each 64-bit polynomial coefficient. We propose a novel and faster approach, called negative double Montgomery reductions, for achieving this task. This technique reduces the instruction counts of double modular reductions to just five. Additionally, we show that the lazy reduction strategy is not only applicable in the NTT/INTT implementation but also significantly reduce the time complexity of four key subroutines in Raccoon. We perform a detailed range analysis to ensure the correctness of the Raccoon implementation with the lazy reduction technique. Furthermore, we optimize several 64-bit polynomial arithmetic operations tailored to Raccoon's parameters, such as left-shift/right-shift, coefficient reduction, and conditional operations. The proposed implementations reduce 32.46% $\sim$ 40.01% of the clock cycles compared to Raccoon's reference implementation. Finally, we explore memory optimization techniques for the lightweight implementations of the speed-optimized Kyber and high-order Raccoon, addressing the challenges posed by memory-constrained IoT devices. These optimizations significantly improve the feasibility and practicality of deploying LBC schemes Kyber and Raccoon, ensuring lightweight quantum-safe protection and side-channel resistance on memory-limited devices.

1.5 Dissertation outline

The outline of this thesis is summarized as follow.

Chapter 1 gives an overall introduction of the background, post-quantum cryptography, the challenges of deploying PQC on IoT devices, cryptographic engineering, and the contributions of this thesis.

Chapter 2 provides a brief introduction of the mathematical basis of LWE, the algorithm descriptions of the LBC schemes, and the core operations underlying these LBC schemes.

Chapter 3 reviews the fundamental and state-of-the-art modular arithmetic employed in LBC schemes. It then presents the improved Plantard arithmetic, which supports modular arithmetic over signed integers in an extended range. This chapter presents correctness proofs from two distinct perspectives and includes detailed comparisons between the improved Plantard arithmetic and the original Plantard arithmetic, the signed Plantard arithmetic proposed in [15], as well as state-of-the-art Montgomery and Barrett arithmetic. These comparisons highlight the excellent merits of the proposed Plantard arithmetic, demonstrating its potential to replace existing modular arithmetic methods in LBC schemes on specific platforms.

Chapter 4 presents the optimized implementations of modular arithmetic, 16-bit NTT in Kyber and NTRU, small NTT and multi-moduli NTT in Dilithium, as well as the Keccak on Cortex-M4, Cortex-M3, and RISC-V platforms. These optimizations significantly enhance the performance of LBC schemes on IoT devices. Additionally, the memory optimizations for the speed-version Kyber are discussed, aimed at improving its feasibility and practicality on memory-constrained IoT devices.

Chapter 5 presents the efficient, lightweight, and secure implementations of the side-channel secure LBC scheme Raccoon.

Chapter 6 concludes the thesis and discusses the possible future work related to the thesis.

Chapter 2

Background

In this chapter, we provide necessary mathematical background for LBC and dissect the fundamental components of LBC schemes.

2.1 Basic definitions and notations

Let f be a function. If f is deterministic, the output of $f(x)$ is assigned to y using $y := f(x)$. If f is randomized, the assignment is denoted as $y \leftarrow f(x)$. For a given probability distribution $\mathcal{D}$, $y \leftarrow \mathcal{D}$ indicates that y is sampled from $\mathcal{D}$. Let q be an integer and n a power of two.

The quotient ring of integers modulo q is denoted as $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. The ring $\mathcal{R}_q$ is defined as $\mathbb{Z}_q[x]/(x^n + 1)$, representing the quotient of the polynomial ring $\mathbb{Z}_q[x]$ by the ideal generated by $(x^n + 1)$. Italic lowercase letters represent scalars, which are elements of $\mathbb{Z}$, $\mathbb{Z}_q$, or $\mathcal{R}_q$. Bold lowercase letters denote vectors, while bold uppercase letters indicate matrices. Let $f \in \mathcal{R}_q$ be a polynomial. Its negation is defined as $-f = -f_0 + \cdots + (-f_{n-1})x^{n-1}$, where each coefficient of f is negated.

Let v be a positive integer. Any integer $x \in \mathbb{Z}$ can be expressed as $x = 2^v \cdot x_{\text{top}} + x_{\text{bot}}$, where $x_{\text{top}} \in \mathbb{Z}$ and $x_{\text{bot}} \in [-2^{v-1}, 2^{v-1} - 1]$. The function $\lfloor \cdot \rceil_v$ is defined as $\lfloor x \rceil_v = \lfloor x/2^v \rceil = x_{\text{top}}$. The function $\lfloor \cdot \rfloor$ denotes the floor operation.

For $z \in \mathbb{Z}$, we define $z \bmod^{\pm} q \in \{-\lfloor \frac{q}{2} \rfloor, -\lfloor \frac{q}{2} \rfloor + 1, \cdots, \lfloor \frac{q-1}{2} \rfloor - 1, \lfloor \frac{q-1}{2} \rfloor\}$ and $z \bmod^{+} q \in \{0, 1, \cdots, q-1\}$ to represent the canonical signed representative and

canonical unsigned representative, respectively. From an implementation perspective, consider a 32-bit signed or unsigned multiplication with the lower 32 bits of the result are obtained. The binary representations of $z \bmod^{\pm} 2^{32}$ and $z \bmod^{+} 2^{32}$ are identical.

2.1.1 Masking

Masking is a powerful technique used to protect LBC schemes against side-channel attacks. It works by splitting sensitive data, such as secret keys or intermediate values, into multiple "shares" and performing cryptographic operations on these shares instead of the original data. This ensures that no individual share reveals meaningful information about the secret, making it significantly harder for attackers to extract sensitive information through side-channel analysis.

Two fundamental masking methods are commonly employed based on the properties of the operations: Boolean masking and arithmetic masking.

- **Boolean masking** splits sensitive data using the XOR operation, making it well-suited for bit-wise operations such as AND and XOR.

- **Arithmetic masking**, on the other hand, splits sensitive data using modular addition, making it ideal for modular operations prevalent in LBC schemes.

However, many LBC schemes include operations that favor either boolean or arithmetic masking, requiring costly mask conversions to convert between the two forms. These conversions significantly increase the overhead of existing side-channel protection methods.

To mitigate this challenge, Raccoon [95] is specifically designed to be masking-friendly by exclusively using arithmetic masking. This design eliminates the need for expensive mask conversions. In Raccoon, a sensitive variable $\mathbf{x}$ in Raccoon is shared across d components as $(\mathbf{x}_i)_{i\in[d]}$, such that $\mathbf{x} = \sum_{i\in[d]} \mathbf{x}_i \bmod q$. A d-shared variable $(\mathbf{x}_i)_{i\in[d]}$ is denoted as $[\![\mathbf{x}]\!]$.

2.2 Lattice-based cryptography

We briefly recall the basis of lattice in this section and refer more details to [93, 78].

2.2.1 Lattice basis

The lattice-based cryptography is built upon the assumed hardness of lattice hard problems. The so-called lattice is defined as in Definition 2.2.1.

Definition 2.2.1 (Lattice)**.** *Given n linearly independent vectors $\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n} \in \mathbb{R}^m$, an n-dimensional lattice $\mathcal{L}$ is defined as*

$$\mathcal{L}(\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}) = \left\{ \sum x_i \mathbf{b_i} | x_i \in \mathbb{Z} \right\}.$$

The vectors $\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}$ form a *basis* of the lattice $\mathcal{L}$. Let $\mathbf{B}$ be the $m \times n$ matrix with each column being $\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}$, then the lattice $\mathcal{L}$ is equivalent to:

$$\mathcal{L}(\mathbf{B}) = \left\{ \sum \mathbf{B}x | x \in \mathbb{Z}^n \right\}.$$

We refer n as the *rank* and m as the dimension of the lattice. The lattice $\mathcal{L}$ is a full-rank lattice when $m = n$, which is a general case in cryptography. The Euclidean length of a vector $\mathbf{v} = (v_1, v_2, \cdots v_n) \in \mathbb{R}^n$ is computed as $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}$. The minimum distance of a lattice $\mathcal{L}$ is defined as the length of the shortest non-zero vector, denoted as $\lambda_1(\mathcal{L}) := \min_{\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\mathbf{v}\|$.

2.2.2 SVP and CVP

There are many variants of lattice computational problems used in lattice-based cryptography. Two of the most fundamental problems are Shortest Vector Problem (SVP) and Closest Vector Problem (CVP) [84]. While these problems are not directly used in modern LBC schemes, their conjectured intractability forms the cornerstone of many lattice-based cryptographic constructions. A brief understanding of these problems is essential for appreciating the security assumptions underlying the lattice-based cryptography.

The SVP, as illustrated in Definition 2.2.2, is defined as the task of finding the shortest non-zero vector in a given lattice. This problem is considered to be computationally hard, particularly in high-dimensional spaces. The CVP, as shown in Definition 2.2.3, asks for the closest lattice vector to a given target vector. In other words, given a lattice and a target vector, the objective is to find the lattice point that is closest to the target. Like the SVP, the CVP is believed to be computationally hard in high dimensions.

Definition 2.2.2 ((Approximate) Shortest Vector Problem (SVP) [84])**.** *Given a basis* $\mathbf{B}$ *of a lattice* $\mathcal{L} = \mathcal{L}(\mathbf{B})$*, the SVP problem asks to find a shortest non-zero vector* $\mathbf{v} \in \mathcal{L}$ *such that* $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$*. For a parameter* $\gamma \geq 1$*, the approximate variant* SVP_γ *asks to find a non-zero vector* $\mathbf{v} \in \mathcal{L}$ *such that* $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L})$*.*

Definition 2.2.3 ((Approximate) Closest Vector Problem (CVP) [84])**.** *Given a basis* $\mathbf{B}$ *of a lattice* $\mathcal{L} = \mathcal{L}(\mathbf{B})$ *and a target vector* $\mathbf{t}$*, the CVP problem ask to find a vector* $\mathbf{v} \in \mathcal{L}$ *such that the distance between* $\mathbf{v}$ *and* $\mathbf{t}$*, i.e.,* $\|\mathbf{v} - \mathbf{t}\|$*, is minimized. For a parameter* $\gamma \geq 1$*, the approximate variant* CVP_γ *asks to find a vector* $\mathbf{v} \in \mathcal{L}$ *such that* $\|\mathbf{v} - \mathbf{t}\| \leq \gamma \cdot \lambda_1(\mathcal{L})$*.*

Although the SVP and CVP are not directly employed in many modern LBC schemes, their intractability is fundamental to the security assumptions of more advanced lattice problems. Specifically, the problems of Learning With Errors (LWE) and Short Integer Solution (SIS), which form the basis for post-quantum cryptographic standards such as Kyber and Dilithium, derive their security from the hardness of certain variants of SVP or CVP problems.

2.2.3 SIS and LWE

In the past decade, two lattice-based computational problems, namely Learning With Errors (LWE) [100] and Short Integer Solution (SIS) [8], have gained prominence in cryptography because they have strong security guarantee by connecting the *worst-case* and *average-case* hardness. Ajtai [8] and Regev [100] demonstrated

that these problems are hard on average, provided that certain related problems remain intractable in the worst case. Therefore, these two hard problems form the foundation of the security of many cryptographic schemes. For example, the FrodoKEM [13], which is based on the LWE problem, is a notable cryptographic protocol built on these principles, offering strong security guarantees. The formal definitions of these problems are provided in Definitions 2.2.4 and 2.2.5.

Definition 2.2.4 (Short Integer Solution Problem (SIS) [8])**.** *Given a uniformly distributed matrix* $\mathbf{A}_q^{n\times m}$*, the SIS problem* $SIS_{q,m,\beta}$ *asks to find a vector* $\mathbf{z} \in \mathbb{Z}^m$ *such that* $\mathbf{Az} = 0 \bmod q$ *and* $0 < \|\mathbf{z}\| \leq \beta$*.*

Definition 2.2.5 (Learning with Errors (LWE) [100])**.** *The search version of LWE problem* $sLWE_{n,m,q,\chi}$ *is defined as follows: Given a uniformly random secret vector* $\mathbf{s}$ *chosen in* $\mathbb{Z}_q^n$*, an error* $e \leftarrow \chi$*, and a uniformly random vector* $\mathbf{a} \in \mathbb{Z}_q^n$*, and given* m *samples of the form* $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \bmod q)$*, the* $sLWE_{n,m,q,\chi}$ *problem asks to recover the secret vector* $\mathbf{s}$*. The decision version, on the other hand, involves distinguishing between two distributions: the distribution of* m *independent samples from* $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \bmod q)$ *and* m *independent samples from a uniform distribution* $U(\mathbb{Z}_q^n \times \mathbb{Z}_q)$*.*

When the number of the requested samples $(\mathbf{a}_i, \langle \mathbf{s}, \mathbf{a}_i \rangle + e_i)$ is m, then we are considering the matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ with each row being $\mathbf{a}_i$ and $\mathbf{e} = (e_1, \cdots, e_m)^T$ in the LWE problem. This can be interpreted in terms of linear algebra similar to the SIS problem. This allows us to express the problem as $(\mathbf{A}, \mathbf{As} + \mathbf{e} = \mathbf{b} \bmod q)$. This setup is similar to the Short Integer Solution (SIS) problem in the sense that both involve finding a vector (in this case, the secret $\mathbf{s}$ from an equation involving a matrix and an error term. The main difference is that the LWE problem involves a noise term $\mathbf{e}$ that is typically sampled from some distribution χ.

2.2.4 RSIS and RLWE

Although cryptographic schemes constructed based on the standard LWE problem offer strong security guarantees, they are often very slow, which hinders their prac-

tical use in real-world applications, especially on resource-constrained IoT devices. For example, the state-of-the-art FrodoKEM implementation on ARM Cortex-M4 [71] is at least 100× slower than our Kyber implementation [63].

Definition 2.2.6 (Ring SIS Problem (RSIS))**.** *Given $a_1, a_2, \cdots, a_m \in R_q$ chosen independently from the uniform distribution, the $RSIS_{q,m,\beta}$ problem asks to find $z_1, z_2, \cdots, z_m \in R$ such that $\sum_{i=1}^{m} a_i \cdot z_i = 0 \bmod q$ and $0 < \|\mathbf{z}\| \leq \beta$, where $\mathbf{z} = (z_1, z_2, \cdots, z_m)^T \in R^m$.*

Definition 2.2.7 (Ring LWE Problem (RLWE) [100])**.** *Given $a_1, a_2, \cdots, a_m \in R_q$ chosen independently from the uniform distribution, the search version of RLWE problem $rRLWE_{m,q,\chi}$ is defined as follows: Let s be the uniformly chosen secret in R_q. Given m samples of $(a_i, a_i \cdot s + e_i \bmod q)$, where the coefficients of e_i are independently sampled from the error distribution χ, the goal of finding s is considered difficult.*

To improve the efficiency of LBC schemes based on SIS and LWE problems, researchers [94, 81, 100] introduced the ring version of SIS and LWE problems by incorporating the ring of integers (i.e., polynomial ring R_q), which has efficient arithmetic properties. These are formalized in Definition 2.2.6 and 2.2.7. Due to their excellent efficiency, the ring version lattice problems have become the foundation of several popular PQC candidates in the NIST PQC standardization project, such as NewHope [14] and qTESLA [11]. However, despite having reductions from worst-case lattice problems, the hardness of these schemes is restricted to ideal lattices due to the introduction of the ring of integers, which may enable more efficient attacks [17]. In particular, the GapSVP problem over ideal lattices is relatively easy, raising concerns regarding their security [78].

2.2.5 MSIS and MLWE

More recently, Langlois and Stehlé [78] proposed the Module SIS (MSIS) and Module LWE (MLWE) problems, which generalize the standard SIS and LWE problems to module lattices. These new problems bridge the connection between SIS and RSIS,

as well as between LWE and RLWE, respectively. They also provided worst-case to average-case reductions for these module problems. The term "module" refers to an algebraic structure that generalizes both rings and vector spaces, while the module lattice generalizes standard lattices and ideal lattices [78].

Let R_q be an n-dimensional polynomial ring, d denotes the rank of the module $M \subseteq R_q^d$, then the dimension of the corresponding module lattice is $N = nd$. The SIS and LWE problems defined over module lattices are illustrated in Definitions 2.2.8 and 2.2.9. The MSIS and MLWE problems can also be interpreted in terms of linear algebra, similar to SIS and LWE problems, where $\mathbf{A}$ is an $d \times m$-dimensional matrix and each entry is a n-dimensional polynomial in the ring R.

Definition 2.2.8 (Module SIS Problem (MSIS))**.** *Given* $\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_m \in R_q^d$ *chosen independently from the uniform distribution, the* $MSIS_{q,m,\beta}$ *problem asks to find* $z_1, z_2, \cdots, z_m \in R$ *such that* $\sum_{i=1}^{m} \mathbf{a}_i \cdot z_i = 0 \bmod q$ *and* $0 < \|\mathbf{z}\| \leq \beta$*, with* $\mathbf{z} = (z_1, z_2, \cdots, z_m)^T \in R^m$*.*

Definition 2.2.9 (Module LWE Problem (MLWE))**.** *Given* $\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_m \in R_q^d$ *chosen independently from the uniform distribution and the coefficients of* $\mathbf{e}$ *being independently sampled from* χ*, the search version* $sMLWE_{q,m,\beta}$ *problem is defined as follows: Let* $\mathbf{s} \in R_q^d$ *be the secret, and given* m *samples of the form* $(\mathbf{a}_i, \mathbf{a}_i \cdot \mathbf{s} + \mathbf{e} \bmod q)$*, finding* $\mathbf{s}$ *is considered hard.*

Two of the NIST PQC finalists, Kyber and Dilithium are constructed based on these module version of lattice problems. Due to the introduction of module lattices, they offer good security, scalability and excellent performance [17, 42]. Thanks to these merits, they are well-recognized by NIST and cryptographic academy, and hence are standardized by NIST as ML-KEM [90] and ML-DSA [91]. Therefore, these two problems will be the main focus in this thesis.

2.3 LBC schemes

We now present the algorithm descriptions of the LBC schemes involved in the thesis.

2.3.1 Kyber

The security of Kyber is based on the MLWE problem, whose formal definition is similar to Definition 2.2.9. In Kyber, the secret vector $\mathbf{s}$ is sampled from a centered binomial distribution $B_\eta(R_q^{k\times 1})$, where k is the dimension of the underlying MLWE problem. The public matrix $\mathbf{A}$ is sampled from a uniform random distribution $\mathcal{U}(R_q^{k\times k})$. The decisional MLWE problem indicates that $(\mathbf{A}, \mathbf{b})$ is computationally indistinguishable from a uniform random sampling.

Algorithm 1 Kyber.PKE KeyGen [31]

Output: $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$

1: $seed \leftarrow \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \texttt{SHAKE256}(64, seed)$
3: $\hat{\mathbf{A}} \leftarrow \texttt{GenMatrixA}(\rho)$
4: $\mathbf{s} \leftarrow \texttt{SampleVec}(\sigma, 0)$
5: $\mathbf{e} \leftarrow \texttt{SampleVec}(\sigma, 1)$
6: $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$
7: **return** $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$

Algorithm 2 Kyber.PKE Decryption [31]

Input: $c = (\mathbf{u}', h)$, $sk = \hat{\mathbf{s}}$

Output: Message $\mu \in R_q$

1: $\mathbf{u} \leftarrow \texttt{Decompress}(\mathbf{u}')$
2: $\mathbf{v}' \leftarrow \texttt{Decompress}(h)$
3: **return** $\mu = \mathbf{v}' - \mathrm{NTT}^{-1}\left(\hat{\mathbf{s}}^T \circ \mathrm{NTT}(\mathbf{u})\right)$

Algorithm 3 Kyber.PKE Encryption [31]

Input: $pk = (\hat{\mathbf{b}}, \rho)$, message $\mu \in R_q$, seed $coin \in \{0, \cdots, 255\}^{32}$

Output: Ciphertext $(\mathbf{u}', h)$

1: $\hat{\mathbf{A}} \leftarrow \texttt{GenMatrixA}(\rho)$
2: $\mathbf{s}' \leftarrow \texttt{SampleVec}(coin, 0)$
3: $\mathbf{e}' \leftarrow \texttt{SampleVec}(coin, 1)$
4: $\mathbf{e}'' \leftarrow \texttt{SampleVec}(coin, 2)$
5: $\hat{\mathbf{t}} \leftarrow \mathrm{NTT}(\mathbf{s}')$
6: $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}\left(\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}\right) + \mathbf{e}'$
7: $\mathbf{v}' \leftarrow \mathrm{NTT}^{-1}\left(\hat{\mathbf{b}}^T \circ \hat{\mathbf{t}}\right) + \mathbf{e}'' + \mu$
8: **return** $(\mathbf{u}' = \texttt{Compress}(\mathbf{u}), h = \texttt{Compress}(\mathbf{v}'))$

The polynomial ring used within Kyber is $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$ and $n = 256$ across all parameter sets. Each polynomial coefficient in R_q can be accommodated in a 16-bit integer, which enables a 16-bit NTT polynomial mul-

tiplication. Kyber constructs a CCA-secure KEM from a CPA-secure public key encryption (PKE). Kyber PKE consists of key generation (Algorithm 1), encryption (Algorithm 3), and decryption (Algorithm 2). We refer readers to Kyber's specification [17] for more details about the CCA-secure KEM. The core operation of Kyber in key generation and encryption is the matrix-vector multiplication, i.e., $\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s})$ and $\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}$, where each element of the matrix and vector is a polynomial.

2.3.2 NTTRU

The original NTRU, which was proposed in 1996 and first published in 1998 [61] by Hoffstein, Pipher, and Silverman, has gone through a long research history. As one of the KEM finalists, NTRU has been significantly improved in terms of security and efficiency compared with the original design.

Algorithm 4 NTTRU.PKE KeyGen [82]

Output: $sk = f, pk = h$

1: $f' \leftarrow \beta_2^{768}$
2: $f := 3f' + 1$
3: if f is not invertible in R_q, restart
4: $g \leftarrow \beta_2^{768}$
5: $h := 3g/f$
6: **return** $(sk = f, pk = h)$

Algorithm 5 NTTRU.PKE Encryption [82]

Input: Message m, randomness r, $pk = h$

Output: Ciphertext c

1: $c := hr + m$
2: **return** c

Algorithm 6 NTTRU.PKE Decryption [82]

Input: Ciphertext c, $sk = f$

Output: Message m

1: $m := \left(cf \bmod^{\pm} q\right) \bmod^{\pm} 3$
2: **return** m

The NTRU submitted to NIST is the combination of two first-round candidates, NTRUEncrypt [118] and NTRU-HRSS-KEM [103]. The KEM specified in the NTRU's specification is constructed using a generic transformation of a correct deterministic public-key encryption scheme (correct DPKE)[34]. The polynomial arithmetic of NTRU operates in three polynomial rings $\mathbb{Z}_3[x]/\Phi_{\mathbf{n}}, \mathbb{Z}_q[x]/\Phi_{\mathbf{n}}$, and $\mathbb{Z}_q[x]/\left(\mathbf{\Phi_1} \cdot \mathbf{\Phi_n}\right)$ with $\Phi_1 = (x-1)$ and $\Phi_{\mathbf{n}} = (x^{n-1} + x^{n-2} + \cdots + 1)$.

NTTRU [82], an NTT-friendly variant of NTRU, is constructed using a partially correct probabilistic public-key encryption scheme (partially correct PPKE). The

speed of NTTRU is faster than other NTRU variants, including the third-round finalist NTRU and alternate candidate NTRU Prime. However, the authors only provided security proof under the classical random oracle model (ROM) instead of the quantum random oracle model (QROM).

The public key encryption scheme of NTTRU consists of key generation (Algorithm 4), encryption (Algorithm 5), and decryption (Algorithm 6). The polynomial ring of NTTRU is $R_q = \mathbb{Z}_q[X]/(X^{768} - X^{384} + 1)$, where $q = 7681$. The β_2^{768} is a binomial distribution, which generates $a_1, a_2, a_3, a_4 \leftarrow \{0,1\}^{768}$, and outputs $a_1 + a_2 - a_3 - a_4 \bmod^{\pm} 3 \in R_q$. The modular reduction ($\bmod^{\pm} q$) maps integers to the range $(-\frac{q}{2}, \frac{q}{2})$. The NTT's suitability for NTTRU comes from its polynomial ring $\mathbb{Z}_{7681}[X]/(X^{768} - X^{384} + 1)$, and we will give more details in Subsection 2.4.1.

2.3.3 Dilithium

Dilithium belongs to the LBC category, and its hardness is based on the MLWE and MSIS problems. Dilithium follows the "Fiat-Shamir with Aborts" approach proposed in [79, 80] to construct the digital signature scheme. The key generation (`keygen`), signature generation (`sign`), and signature verification (`verify`) of Dilithium are shown in Algorithm 7, Algorithm 9, and Algorithm 8, respectively. Due to the introduction of the module structure, Dilithium needs to handle a large $k \times l$-dimensional matrix $\mathbf{A}$, where each entry is a degree-$(n-1)$ polynomial over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$ with modulus $q = 2^{23} - 2^{13} + 1 = 8380417$. The module structure brings excellent flexibility to the implementation of Dilithium, which allows us to reuse the fundamental component for different parameter sets of Dilithium.

As shown in Algorithm 7∼8, Dilithium extensively uses SHA-3 or SHAKE primitives for expanding the matrix $\mathbf{A}$ (ExpandA), secret vector $\mathbf{s}_1$ and $\mathbf{s}_2$ (ExpandS), vector y (ExpandMask), small polynomial c with τ non-zero coefficients ± 1's (SampleInBall) and other hashing (H). The MakeHint, UseHint, Power2Round functions are used to drop the lower d bits of $\mathbf{t}$ and reduce the public key size. The parame-

ters used in these algorithms are shown in Table 5.1. For details of these functions and parameters, we refer interested readers to the specification of Dilithium [42].

Algorithm 7 Dilithium key generation (`keygen`) [42]

Output: $pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := \mathrm{H}(\zeta)$
3: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
4: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \texttt{ExpandS}\,(\rho')$
5: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) := \mathrm{Power2Round}_q(\mathbf{t}, d)$
7: $tr \in \{0,1\}^{256} := \mathrm{H}\,(\rho \| \mathbf{t}_1)$
8: **return** $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

Algorithm 8 Dilithium signature verification (`verify`) [42]

Input: Public key pk, message M, and signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

Output: Accept or reject

1: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{512} := \mathrm{H}\,(\mathrm{H}\,(\rho \| \mathbf{t}_1) \| M)$
3: $c := \texttt{SamplelnBall}(\tilde{c})$
4: $\mathbf{w}_1' := \texttt{UseHint}_q\left(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2\right)$
5: **return** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![\tilde{c} = \mathrm{H}\,(\mu \| \mathbf{w}_1')]\!]$ **and** $[\![\#\text{ of 1 's in } \mathbf{h} \text{ is } \leq \omega]\!]$

Algorithm 9 Dilithium signature generation (`sign`) [42]

Input: Secret key sk and message M

Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

1: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{512} := \mathrm{H}(tr \| M)$
3: $\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
4: $\rho' \in \{0,1\}^{512} := \mathrm{H}(K \| \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)
5: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ do
6: $\quad \mathbf{y} \in \tilde{S}_{\gamma_1}^\ell := \texttt{ExpandMask}\,(\rho', \kappa)$
7: $\quad \mathbf{w} := \mathbf{A}\mathbf{y}$
8: $\quad \mathbf{w}_1 := \texttt{HighBits}_q\,(\mathbf{w}, 2\gamma_2)$
9: $\quad \tilde{c} \in \{0,1\}^{256} := \mathrm{H}\,(\mu \| \mathbf{w}_1)$
10: $\quad c \in B_\tau :=$ SamplelnBall $(\tilde{c})$
11: $\quad \mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
12: $\quad \mathbf{r}_0 := \texttt{LowBits}_q\,(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
13: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z}, \mathbf{h}) := \perp$
14: $\quad$ **else**
15: $\quad\quad \mathbf{h} := \mathrm{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$
16: $\quad\quad$ **if** $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$
17: $\quad \kappa := \kappa + \ell$
18: **return** $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

2.3.4 Raccoon

Raccoon's core functions include key generation (Algorithm 10), signing (Algo-

Table 2.1: Dilithium parameters [42]

NIST security level	2	3	5
q [modulus]	8380417	8380417	8380417
n [the order of polynomial]	256	256	256
d [drop bits from $\mathbf{t}$]	13	13	13
τ [# of ± 1's in c]	39	49	60
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, l) [dimensions of $\mathbf{A}$]	(4,4)	(6,5)	(8,7)
η [secret key range]	2	4	2
$\beta = \tau \cdot \eta$ [$c\mathbf{s}_i$ coefficient range]	78	196	120
$\mathbf{t}_0$ coefficient range	2^{12}	2^{12}	2^{12}
$\mathbf{t}_1$ coefficient range	2^{10}	2^{10}	2^{10}

rithm 12), and verification (Algorithm 11). Both key generation and signing are implemented in masked form, with an unmasked version available when $d = 1$.

In the `keygen` process, d-shared small errors $[\![s]\!]$ and $[\![e]\!]$ are generated in lines 4 and 6, respectively. The verification key is computed as an LWE sample $(\mathbf{A}, \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$.

Raccoon follows the Fiat-Shamir paradigm, with the `sign` process notably avoiding rejection sampling. The main steps are as follows: (1) Commit (lines 4 to 9): Random noise is generated in masked form, and the LWE commitment $\mathbf{w}$ is computed in masked form. Afterward, $\mathbf{w}$ is unmasked. (2) Challenge (lines 10 and 11): A challenge is derived using the message msg, the verification key vk, and the unmasked LWE commitment $\mathbf{w}$. (3) Response (lines 14 to 18): A response $(\mathbf{h}, \mathbf{z})$ is computed based on the random noise, the private key, and the challenge. The second part of the response, $\mathbf{z}$, is unmasked since it only involves public data.

The `verify` process aligns closely with other lattice-based Fiat-Shamir signatures, following standard procedures.

Algorithm 10 Raccoon key generation (`keygen`) [95]

Input: $\emptyset$

Output: vk, sk

1: $\mathsf{seed} \leftarrow \{0,1\}^{\kappa}$
2: $\mathbf{A} := \texttt{ExpandA}(\mathsf{seed})$
3: $[\![\mathbf{s}]\!] \leftarrow l \times \texttt{ZeroEncoding}(d)$
4: $[\![\mathbf{s}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{s}]\!], u_t, \texttt{rep})$
5: $[\![\mathbf{t}]\!] := \mathbf{A} \cdot [\![\mathbf{s}]\!]$
6: $[\![\mathbf{t}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{t}]\!], u_t, \texttt{rep})$
7: $\mathbf{t} := \texttt{Decode}([\![\mathbf{t}]\!])$
8: $\mathbf{t} := \lfloor \mathbf{t} \rceil_{v_t}$
9: **return** $(\mathsf{vk} = (\mathsf{seed}, \mathbf{t}), \mathsf{sk} = (\mathsf{vk}, [\![\mathbf{s}]\!]))$

Algorithm 11 Raccoon signature verification (`verify`) [95]

Input: $\mathsf{vk} = (\mathsf{seed}, \mathbf{t})$, msg, and $\mathsf{sig} = (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$

Output: OK (accept) or FAIL (reject).

1: $(c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z}) := \mathsf{sig}, (\mathsf{seed}, \mathsf{t}) := \mathsf{vk}$
2: **if** $\{\texttt{CheckBounds}(\mathsf{sig}) = \mathsf{FAIL}\}$ **then** FAIL
3: $\mu := \mathrm{H}(\mathrm{H}(\mathsf{vk})||\mathsf{msg})$
4: $\mathbf{A} := \texttt{ExpandA}(\mathsf{seed})$
5: $c_{\mathsf{poly}} := \texttt{ChalPoly}(c_{\mathsf{hash}})$
6: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - 2^{v_t} \cdot c_{\mathsf{poly}} \cdot \mathbf{t}$
7: $\mathbf{w}' := \lfloor \mathbf{y} \rceil_{v_w} + \mathbf{h}$
8: $c'_{\mathsf{hash}} := \texttt{ChalHash}(\mathbf{w}', \mu)$
9: **if** $c_{\mathsf{hash}} \neq c'_{\mathsf{hash}}$ **return** FAIL
10: **return** OK

Algorithm 12 Raccoon signature generation (`sign`) [95]

Input: $\mathsf{sk} = (\mathsf{vk}, [\![\mathbf{s}]\!])$ and $\mathsf{msg} \in \{0,1\}^*$.

Output: Signature $\mathsf{sig} = (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$

1: $(\mathsf{vk}, [\![\mathbf{s}]\!]) := \mathsf{sk}, (\mathsf{seed}, \mathsf{t}) := \mathsf{vk}$
2: $\mu := \mathrm{H}(\mathrm{H}(\mathsf{vk})||\mathsf{msg})$
3: $\mathbf{A} := \texttt{ExpandA}(\mathsf{seed})$
4: $[\![\mathbf{r}]\!] \leftarrow l \times \texttt{ZeroEncoding}(d)$
5: $[\![\mathbf{r}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{r}]\!], u_w, \texttt{rep})$
6: $[\![\mathbf{w}]\!] := \mathbf{A} \cdot [\![\mathbf{r}]\!]$
7: $[\![\mathbf{w}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{w}]\!], u_w, \texttt{rep})$
8: $\mathbf{w} := \texttt{Decode}([\![\mathbf{w}]\!])$
9: $\mathbf{w} := \lfloor \mathbf{w} \rceil_{v_w}$
10: $c_{\mathsf{hash}} := \texttt{ChalHash}(\mathbf{w}, \mu)$
11: $c_{\mathsf{poly}} := \texttt{ChalPoly}(c_{\mathsf{hash}})$
12: $[\![\mathbf{s}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{s}]\!])$
13: $[\![\mathbf{r}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{r}]\!])$
14: $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$
15: $[\![\mathbf{z}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{z}]\!])$
16: $\mathbf{z} := \texttt{Decode}([\![\mathbf{z}]\!])$
17: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - 2^{v_t} \cdot c_{\mathsf{poly}} \cdot \mathbf{t}$
18: $\mathbf{h} := \mathbf{w} - \lfloor \mathbf{y} \rceil_{v_w}$
19: $\mathsf{sig} := (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$
20: **if** $\{\texttt{CheckBounds}(\mathsf{sig}) = \mathsf{FAIL}\}$ **goto** Line 4
21: **return** sig

The $[\![z]\!] \leftarrow \texttt{ZeroEncoding}(d)$ subroutine takes as input a power-of-two integer d and outputs a uniform d-sharing $[\![z]\!] \in \mathcal{R}_q^d$ of $0 \in \mathcal{R}_q$, denoted as $[\![0]\!]$. The $[\![x]\!]' \leftarrow$

`Refresh`($[\![x]\!]$) subroutine takes as input a d-sharing $[\![x]\!]$ of $x \in \mathcal{R}_q$, refreshes the input d-sharing, and outputs a fresh d-sharing satisfying $[\![x]\!]' = [\![x]\!] + [\![0]\!]$. $\texttt{SU}(u, T) = [T] \cdot \mathcal{U}(-2^{u-1}, \ldots, 2^{u-1} - 1)$ represents the sum of T independent uniform random variables, each sampled from the uniform distribution $\mathcal{U}(-2^{u-1}, \ldots, 2^{u-1} - 1)$. The $[\![v]\!] \leftarrow \texttt{AddRepNoise}([\![v]\!], u, \texttt{rep})$ subroutine implements $\texttt{SU}(u, d \cdot \texttt{rep})$. For each masked coefficient $[\![a]\!]$ of $[\![v]\!]$, every share of $[\![a]\!]$ is perturbed by adding small uniform noise, then $[\![a]\!]$ is refreshed. This process is repeated `rep` times. For details on other auxiliary functions used in Raccoon, we refer to [95].

2.3.5 Core operations of LBC

We now analyze the core operations in Lattice-Based Cryptography (LBC) to highlight the optimization targets of this thesis.

The first core operation is the polynomial multiplication over polynomial ring R_q. The polynomial multiplication in LBC schemes can typically be accelerated using the Number Theoretic Transform (NTT), which reduces the complexity of polynomial multiplication from $O(n^2)$ down to $O(n \log n)$. The NTT computation has been widely studied and optimized in various works [104, 32, 12, 3, 57, 2, 23]. The optimizations normally come from exploring the optimized implementation of modular arithmetic utilizing the avaliable instructions of the target platforms, reducing the memory access using the layer merging strategy, minimizing the modular reductions, and optimizing the computation process with proper precomputation of the twiddle factors in the NTT/INTT computation. In this thesis, we revisit these optimization techniques and propose further enhancements from both an algorithmic and an implementation perspective to improve the performance of NTT-based polynomial multiplication in LBC schemes.

Another critical operation is the sampling of polynomials, secrets, and errors in LBC schemes. This step requires a large amount of pseudorandom numbers for sampling polynomial vectors and matrices. The pseudorandom number generation (PRNG) in these schemes is typically implemented using symmetric primitives, es-

pecially the SHA-3 and SHAKE families, which rely on the Keccak permutation. The efficiency of Keccak is therefore central to the overall performance of LBC schemes. As observed in pqm4 [71], the Keccak permutation can account for over 70% of the running time in optimized implementations of Kyber and Dilithium. Therefore, optimizing Keccak's efficiency is crucial for the practical deployment of LBC schemes in real-world applications, particularly on resource-constrained platforms like IoT devices. This thesis will also explore optimizations for Keccak to ensure more efficient LBC implementations.

2.4 Polynomial multiplication

In this thesis, NTT is classified into 16-bit NTT [82, 17], 32-bit NTT [36, 115], or 64-bit NTT [95] according to the modulus size. For example, the modulus size of Kyber and NTTRU are both smaller than 2^{16}, so the NTT of these moduli is classified into the 16-bit NTT. The modulus of Dilithium is $q = 8380417$, which is larger than 16-bit and smaller than 23-bit; thus, the NTT of modulus $q = 8380417$ is classified into the 32-bit NTT. In the case of the polynomial multiplication in Raccoon, which by default uses 64-bit NTT, an optimization strategy for deploying the 64-bit NTT efficiently on 32-bit platforms involves splitting the 64-bit NTT into two 32-bit NTTs using the Chinese Remainder Theorem (CRT).

2.4.1 NTT in Kyber and NTTRU

NTT multiplication over the polynomial ring $\mathbb{Z}_q[X]/f(X)$ is based on the fact that the polynomial $f(X)$ can be factored as

$$f(X) = \prod_{i=0}^{n-1} f_i(X) (\mathrm{mod}\, q),$$

where $f_i(X)$ are small-degree (degree-0 for a complete-NTT, higher degree for an incomplete-NTT [36]) polynomials. The polynomial multiplication of $a, b \in \mathbb{Z}_q[X]/f(X)$ is first calculated using NTT as

$$a_i = a \bmod f_i(X) \text{ and } b_i = b \bmod f_i(X) \tag{2.4.1}$$

for $i = 0, \cdots, n-1$. Then we compute the base multiplication as $a_0 b_0, \cdots, a_{n-1} b_{n-1}$. Finally, using the inverse NTT (INTT), we find the result polynomial c such that $c = ab \bmod f(X)$.

For Kyber, the polynomial $f(X) = X^{256} + 1$ can be factored, with the help of a primitive 256-th root of unity ζ, as

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \zeta^{2i+1}\right).$$

This factorization is incomplete since $f(X)$ is factored as degree-1 polynomials instead of degree-0. Therefore, the NTT of Kyber is a 7-layer incomplete-NTT.

For NTTRU, the polynomial $f(X) = X^{768} - X^{384} + 1$ is initially split into $(X^{384} + 684)(X^{384} - 685)$, then all the way down to irreducible polynomials $X^3 \pm r$. The NTT of NTTRU consists of 8 layers of butterflies, and we refer readers to [82] for more details.

The commonly used computing structures of NTT are Cooley-Tukey (CT) butterfly [37] and Gentleman-Sande (GS) butterfly [51]. The CT butterfly accepts inputs in normal order and generates outputs in bit-reverse order. In contrast, the GS butterfly takes inputs in bit-reverse order and produces outputs in normal order. Therefore, a hybrid structure which uses CT butterfly for NTT and GS butterfly for INTT will accept inputs and generate outputs both in normal order without any bit-reversal operations.

2.4.2 NTT in Dilithium

The polynomial multiplication of Dilithium is performed over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 8380417$ and $n = 256$. The modulus q satisfies $q \equiv 1 \bmod 2n$ so that a $2n$-th root of unity ($\zeta = 1753$) exists. Since $\zeta^{256} = -1 \bmod q$, one can then split the cyclotomic polynomial as $X^{256} + 1 = X^{256} - \zeta^{256} = (X^{128} - \zeta^{128})(X^{128} + \zeta^{128})$ using the NTT, which is a variant of the fast Fourier transform

(FFT) over a finite field. This factorization process can be iteratively continued for the two degree-128 polynomials separately. Note that $X^{128} + \zeta^{128}$ is equivalent to $X^{128} - \zeta^{384}$ and it can be further factorized as $X^{128} - \zeta^{384} = (X^{64} - \zeta^{192})(X^{64} + \zeta^{192})$. The parameters of Dilithium ($q \equiv 1 \bmod 2n$) enable us to perform a complete 8-layer NTT (complete NTT) and factorize the cyclotomic polynomial $X^n + 1$ into linear factors $X - \zeta^i$ with $i = 1, 3, 5, \ldots, 511$. (Note that there are also incomplete NTTs where we can only factorize the cyclotomic polynomial into factors $X^2 - \zeta^i$; see Kyber [31] for example.) The cyclotomic polynomial ring R_q is isomorphic to the product of the rings $\mathbb{Z}_q[X]/(X - \zeta^i)$ according to the Chinese remainder theorem (CRT):

$$a \mapsto \left(a(\zeta), a\left(\zeta^3\right), \ldots, a\left(\zeta^{511}\right)\right) : R_q \to \prod_i \mathbb{Z}_q[X]/\left(X - \zeta^i\right).$$

After the NTT transform, the degree-$(n-1)$ polynomial multiplication of a and b is transformed into n computationally inexpensive pointwise multiplications over NTT-domain $\hat{a}$ and $\hat{b}$. After the pointwise multiplications, the inverse NTT transform (INTT) would convert the NTT-domain elements back to the normal domain. Overall, the time complexity of the polynomial multiplication is reduced down to $O(n \log(n))$, and the polynomial multiplication ab using NTT can be described as INTT(NTT(a)$\circ$ NTT(b)).

2.4.3 Small polynomial multiplications in Dilithium

Because the modulus of Dilithium q is a 23-bit prime number, most of the polynomial multiplications over the polynomial ring R_q are implemented with 32-bit NTT. However, there is a special small polynomial c which has exactly τ ± 1's and $256 - \tau$ 0's. There are four small polynomial multiplications involving c in Algorithm 9 and Algorithm 8, namely $c\mathbf{s}_1, c\mathbf{s}_2, c\mathbf{t}_0,$ and $c\mathbf{t}_1$. For simplicity, we denote $c\mathbf{s}_1$ and $c\mathbf{s}_2$ as $c\mathbf{s}_i$ for $i = 1, 2$, and $c\mathbf{t}_0$ and $c\mathbf{t}_1$ as $c\mathbf{t}_i$ for $i = 0, 1$. Due to the special structure of c, these polynomial multiplications would normally generate a polynomial product with a relatively small coefficient range. For example, the coefficient range of $\mathbf{s}_i$ is $[-\eta, \eta]$, then the coefficients of the product $c\mathbf{s}_i$ are smaller than $\tau \cdot \eta$, denoted as

$\beta = \tau \cdot \eta$. As for $c\mathbf{t}_i$, since the coefficients of $\mathbf{t}_0$ and $\mathbf{t}_1$ are smaller than 2^{10} and 2^{12}, the coefficients of the products $c\mathbf{t}_0$ and $c\mathbf{t}_1$ are smaller than $\tau \cdot 2^{10}$ and $\tau \cdot 2^{12}$ (denoted as $\beta' = \tau \cdot 2^{10}$ or $\beta' = \tau \cdot 2^{12}$), respectively.

According to [36, Section 2.4.6], these kinds of polynomial multiplications can be treated as multiplications over $\mathbb{Z}_{q'}/[X](X^n + 1)$ with a prime modulus $q' > 2\beta$ or $q' > 2\beta'$. Therefore, instead of using 32-bit NTT over R_q for computing $c\mathbf{s}_i$ and $c\mathbf{t}_i$ as in the previous implementation [57], recent research [3] shows that the polynomial multiplications of $c\mathbf{s}_i$ can be implemented with smaller and faster 16-bit NTT by taking advantage of the fact that 2β is smaller than 2^{16}. More precisely, it suggests to use Fermat number transform (FNT) with Fermat number $q' = 257$ or 16-bit NTT with $q' = 769$ for computing $c\mathbf{s}_i$ in different parameter sets. As for $c\mathbf{t}_i$, since the modulus q' that satisfies $q' > 2\beta'$ is larger than 2^{16}, it cannot be accelerated with the faster 16-bit NTT. However, as we will show later, one can still choose a modulus smaller than the q of Dilithium to further speed up $c\mathbf{t}_i$ using the multi-moduli NTT technique on Cortex-M3.

2.4.4 The 16-bit NTT and 32-bit NTT

It should be noted that the constant-time 32-bit NTT is at least 2× slower than the constant-time 16-bit NTT on platforms like Cortex-M3 and AVX2. This is mainly due to the inherent architectural design of these platforms. More specifically, since Cortex-M3 does not have constant-time full multiplications like `UMULL, SMULL, UMLAL`, and `SMLAL`, the constant-time 32-bit modular multiplication proposed in [57] is implemented using the schoolbook multiplication over two 16-bit limbs and is more expensive than 16-bit modular multiplication on Cortex-M3. As a result, the constant-time 32-bit NTT is about 3 times slower than the constant-time 16-bit NTT, and the variable-time 32-bit NTT is also 1.78× slower than the constant-time 16-bit NTT on Cortex-M3 according to [57, Table 2]. This suggests that 16-bit NTT is a more favorable choice over 32-bit NTT on Cortex-M3. Apart from M3, AVX2 is another platform where 16-bit NTT is preferable to 32-bit NTT.

The AVX2 instructions have the capability to process operations simultaneously over sixteen 16-bit coefficients or eight 32-bit coefficients when handling 16-bit NTT or 32-bit NTT. Employing the 32-bit NTT inherently diminishes the parallelism utilization of AVX2, resulting in decreased performance compared to the 16-bit NTT. According to the AVX2 reference implementations of Kyber[1] and Dilithium[2], the 16-bit NTT implementation of Kyber outperforms the 32-bit NTT implementation of Dilithium on AVX2 by a factor of 4.11×. To summarize, the architectural characteristics of Cortex-M3 and AVX2 lead to the constant-time 32-bit NTT being at least 2× slower than the constant-time 16-bit NTT.

2.4.5 Multi-moduli NTT and the explicit CRT

The slower 32-bit NTT on AVX2 motivates Chung et al. [36] to replace the 32-bit NTT with multiple efficient 16-bit NTTs (multi-moduli NTT) with the help of the explicit CRT. Similar to [36], Abdulrahman et al. [2] further demonstrate the feasibility of using multi-moduli NTT for Saber on Cortex-M3 due to the much slower 32-bit NTT than 16-bit NTT. Nevertheless, the multi-moduli NTT technique has not been applied to Dilithium yet. It should be noted that Greconici et al. [57] also briefly discussed the potential to use the multi-moduli NTT and CRT to replace the 32-bit NTT in Dilithium. However, they [57] claimed that the multi-moduli NTT implementation with Montgomery arithmetic was slower than re-implementing the 32-bit multiplication with constant-time 16-bit schoolbook multiplication [57, Section 4.1]. In this paper, we will revisit this idea and demonstrate that by leveraging the efficient Plantard arithmetic for 16-bit NTTs, the multi-moduli NTT remains a viable option for Dilithium on Cortex-M3.

The so-called multi-moduli NTT method [36, 2] chooses s coprime NTT-friendly primes $q_i (i = 0, 1, \ldots, s-1)$ whose product $q' = \prod_{i-0}^{s-1} q_i$ is larger than $2\beta'$. The polynomial multiplication over the composite modulus q' can be carried out by computing NTTs modulo each prime q_i. Recall that the polynomial multiplication of

[1] `https://github.com/pq-crystals/kyber`

[2] `https://github.com/pq-crystals/dilithium`

$c\mathbf{t}_i$ would produce a polynomial with coefficients smaller than a maximum value $\beta' = \tau 2^{10}$ or $\beta' = \tau 2^{12}$. According to [36, Section 2.4.6], we can choose an NTT-friendly prime $q' > 2\beta'$ such that the polynomial multiplication of $c\mathbf{t}_i$ can be implemented with the NTT over q'. Since $2\beta' > 2^{16}$, the polynomial multiplication of $c\mathbf{t}_i$ cannot be accelerated with the 16-bit NTT. Therefore, instead of using the extremely slow 32-bit NTT for $c\mathbf{t}_i$ on Cortex-M3, we can also choose a composite modulus $q' = q_0 q_1$ and utilize the multi-moduli NTT to speed up $c\mathbf{t}_i$. By leveraging the efficient Plantard arithmetic for 16-bit NTTs, we can further optimize the multi-moduli NTT and make it practical for Dilithium on Cortex-M3.

After performing NTTs, pointwise multiplications and INTTs modulo each prime q_i, we can then reconstruct the final results using the explicit CRT. One can either use the Lagrangian interpolation CRT algorithm proposed in [87, Section 4] or the divided-difference interpolation CRT algorithm presented in [36, Theorem 1]. It is suggested that the latter one is better when s is smaller. Because the coefficients produced in $c\mathbf{t}_i$ are quite small, we only need two primes ($s = 2$) for the multi-moduli NTT implementation; so the latter one is best suited for our case. Let q_0, q_1 be coprime ($\gcd(q_0, q_1)$=1), q be the modulus of Dilithium, and $m_1 = q_0^{-1} \bmod^{\pm} q_1$. Let $u \equiv u_i \bmod q_i, i = 0, 1$, where $|u_i| < q_i/2, |u| < q_0 q_1/2$, then the explicit solution for u and $u \bmod^{\pm} q$ is given by $u = u_0 + \big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big)\, q_0$ and $u \bmod^{\pm} q = \big(u_0 + \big(\big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big) \bmod^{\pm} q\big) \cdot q_0\big) \bmod^{\pm} q$, respectively.

2.4.6 Polynomial multiplication in Raccoon

The polynomial ring used in Raccoon is $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where $q = q_1 \times q_2$, $q_1 = 2^{24} - 2^{18} + 1$, $q_2 = 2^{25} - 2^{18} + 1$, and $n = 512$. To enable efficient polynomial multiplication on 32-bit platforms such as the Cortex-M4, the CRT Theorem is employed, similar to previous works [2, 60, 62]. The CRT divides the 49-bit NTT into two smaller NTTs possessing 24-bit and 25-bit moduli respectively. This approach is referred to as multi-moduli NTT in in [2] and can be expressed as an isomorphism $\mathbb{Z}_q \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1}$. This decomposition enables efficient implementation of the 24- and

25-bit NTTs using 32-bit multipliers on 32-bit platforms. The polynomial rings for the 24- and 25-bit NTTs in Raccoon are abbreviated as R_{q_i} for $i = 1, 2$.

Since $q_i - 1$ is divisible by $2n$, there exists a primitive $2n$-th root in $\mathbb{Z}_{q_i}$, which we denote as ζ_{q_i}, where $i \in \{1, 2\}$. The forward NTT involves the following mapping: $\mathbb{Z}_{q_i}[X]/(X^{512} + 1) \cong \prod_{j=0}^{511} \mathbb{Z}_{q_i}[X]/(X - \zeta_{q_i}^{2j+1})$. The inverse NTT (INTT) is the inverse of the above mapping. The NTT-based polynomial multiplication can be expressed as $fg = \text{INTT}(\text{NTT}(f) \odot \text{NTT}(g))$, where $\odot$ represents degree-0 pointwise multiplication. The Cooley-Tukey (CT) butterfly algorithm [37] is often used to implement the forward NTT, while the Gentleman-Sande (GS) butterfly algorithm [51] is frequently employed to implement the INTT. It is pointed out in [2] that using the CT algorithm to implement the INTT also has advantages in certain situations.

The overall process of utilizing the multi-moduli NTT in Raccoon consists of three main steps:

1. Polynomial splitting: Two consecutive modular reductions are performed over q_1 and q_2. For example, for a polynomial coefficient a stored in a 64-bit data type, after two modular reductions, $a \bmod q_1$ and $a \bmod q_2$ directly overwrite the storage space of a. This in-place approach is memory-friendly and preferable to embedded devices.

2. NTT operations: Perform NTT, pointwise multiplications and INTT over q_1 and q_2, respectively.

3. Reconstruction using CRT: Reconstruct the 32-bit results (e.g., $c_1 = c \bmod q_1, c_2 = c \bmod q_2$) modular q using the CRT theorem: $c = c_1 q_2 (q_2^{-1} \bmod q_1) + c_2 q_1 (q_1^{-1} \bmod q_2) \bmod q$.

2.5 Keccak permutation

Keccak is built upon the sponge construction together with a cryptographic permutation Keccak-p$[b, n_r]$ where b and n_r refer to the bit-width of the permutation and the number of rounds, respectively. In this paper, we focus exclusively on instances

where $b = 1600$, such as the variant Keccak-p[1600,24] used in the NIST standards. The state A is represented as an array of 5×5 lanes, each composed of $w = b/25$ bits. $A[x, y]$ refers to the lane at position (x, y) and $A[x, y, z]$ refers to the z-th bit of the lane. Keccak-p is an iterated permutation where each round consists of five consecutive operations θ, ρ, π, χ and ι, where χ is the only non-linear operation. See Listing 2.1 for a brief description of Keccak-p using pseudo-code.

```
# b refers to the permutation width while nr refers to the number of rounds
keccak-p[b,nr](A):
  A = roundperm(A,RC[i])                                          for i in 0..nr-1
  return A

# r[x,y] refer to rotation offsets while RC refers to the round constant
roundperm(A,RC):
  # theta step
  C[x]   = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]     for x in 0..4
  D[x]   = C[x-1] xor rot(C[x+1],1)                               for x in 0..4
  A[x,y] = A[x,y] xor D[x]                                  for (x,y) in (0..4,0..4)
  # rho and pi step
  B[y,2*x+3*y] = rot(A[x,y], r[x,y])                        for (x,y) in (0..4,0..4)
  # chi step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y])         for (x,y) in (0..4,0..4)
  # iota step
  A[0,0] = A[0,0] xor RC
  return A
```

Listing 2.1: Pseudo-code of the Keccak-p cryptographic permutation.

2.6 Conclusion

This section briefly introduces the mathematical basis of LWE, the algorithm descriptions of the LBC schemes, and the core operations underlying these LBC schemes.

Chapter 3

Improved Plantard Arithmetic

This chapter presents the improved Plantard arithmetic introduced in our previous works in 2022 [63] and 2024 [65]. To provide a comprehensive understanding of Plantard arithmetic, we also discuss a concurrent approach proposed by Aoki et al. in 2022 [15], which introduces an alternative variant of signed Plantard arithmetic. We then present an improvement to this variant of Plantard arithmetic in Subsection 3.5.1, which is proposed in our co-authored work [116]. Additionally, we offer a distinct interpretation and correctness proof following the CRT framework proposed by Yang et al. in 2024 [113], which further strengthens the theoretical foundation of Plantard arihtmetic. Note that the correctness proof in this Section 3.6 differs slightly from that of [113].

3.1 Introduction

Modular arithmetic in $\mathbb{Z}_q$ primarily consists of modular addition, modular subtraction, and modular multiplication. These operations can be divided into two stages: the arithmetic operation (addition, subtraction, or multiplication) and the subsequent modular reduction. Modular addition and subtraction are relatively straightforward. The process involves performing the addition or subtraction first, followed by a single subtraction or addition of q, depending on whether the intermediate result exceeds q or falls below 0. In contrast, modular multiplication, which in-

volves two l-bit integers, requires reducing a $2l$-bit product modolo q. The modular reduction in this case is more complex than that for addition or subtraction.

Modern computer architectures typically implement the modulo operation % as $a\%q = a - \lfloor a/q \rfloor \times q$. However, this operation is not constant-time, as its execution time depends on factors such as the values of the dividend a and divisor q, the system architecture, and the specific implementation. The non-constant-time nature of the modulo operation (%) makes it unsuitable for cryptographic applications, as it may leak sensitive information through side-channel attacks. These attacks exploit variations in timing, power consumption, or other physical properties to extract confidential data, such as cryptographic keys or plaintext [77]. Additionally, the final correction step in modular addition and subtraction must also be implemented in a constant-time manner to prevent similar vulnerabilities to side-channel attacks. In summary, modular reduction must be carefully designed for cryptographic implementations. Employing constant-time algorithms for these operations is essential to mitigate side-channel risks and ensure a secure cryptographic implementation.

Modular multiplication is a critical operation within the Number Theoretic Transform (NTT), and the most widely used constant-time modular multiplication algorithms are Montgomery multiplication [86] and Barrett multiplication [19] from 1980s. These algorithms were initially designed for large-moduli cryptosystems such as RSA and ECC. However, the moduli used in lattice-based cryptography (LBC) schemes, including those in the NIST LBC candidates, are relatively small. For instance, Kyber uses a modulus of 3329 [17], NewHope uses 12289 [14], and NTRU employs 7681 [82]. In these cases, the product of two polynomial coefficients typically does not exceed the size of a word (32 or 64 bits). Given the smaller size of these moduli, a more efficient modular multiplication method can be employed. Thomas Plantard proposed a specialized modular multiplication algorithm tailored to the word size of these moduli, referred to as Plantard multiplication [96]. Plantard multiplication optimizes the multiplication of a polynomial coefficient by a constant, leveraging an $l \times 2l$-bit multiplication operation, where l corresponds

to the word size (typically 16 or 32 bits). Notably, if the $l \times 2l$-bit multiplication operation can be executed in a single instruction on the target platform, Plantard multiplication by a constant can reduce the number of multiplication operations by one compared to both Montgomery and Barrett multiplication. This reduction in operations makes Plantard multiplication particularly well-suited for LBC schemes, where performance efficiency is critical.

Motivation. Plantard [96] purely focused on the theoretical design yet leaves much room for further exploration. The original Plantard arithmetic cannot be efficiently applied to the existing LBC schemes for the following reasons. First, the original Plantard multiplication ([96, Algorithm 8]) only presents a solution for multiplying unsigned integers mod an odd modulus q. Previous literature [104, 32, 12, 57, 3] has shown that using unsigned integers in the NTT implementation of LBC schemes is inefficient because it would introduce an extra addition into each butterfly unit. Second, the input range of the original Plantard multiplication is mere $[0, q]$, which requires expensive modular reductions on the polynomial coefficients after each NTT layer.

Previous reports have offered solutions to these problems by adapting the Montgomery arithmetic. Alkim et al. [14] showed that the large input range of Montgomery reduction enables the so-called lazy reduction technique to reduce the use of expensive modular reduction. Seiler et al. [104] proposed a signed version of Montgomery reduction on the Intel platform, which further speeds up the butterfly units. The follow-up work [32, 12, 57] further extended the signed Montgomery reduction to the Cortex-M4 platform. The fastest Montgomery reduction for 16-/32-bit moduli [12, 57, 3] only consumes 2 cycles on the Cortex-M4 platform. Even though Montgomery arithmetic could provide an efficient solution to most of the LBC schemes, this chapter shows that Plantard arithmetic could be a potential alternative to Montgomery arithmetic in the LBC schemes with l-bit odd modulus, given that the target platform could implement the $l \times 2l$-bit multiplication in one multiplication.

Contributions. This chapter aims to improve and customize the Plantard arithmetic for the existing LBC schemes. Our contributions are as follows:

- First, we present an improved Plantard arithmetic for signed integers with its correctness proof. The proposed method is inspired by the observation that the moduli in the existing LBC schemes are normally several times smaller than the modulus constraint in the original Plantard arithmetic.

- We further enlarge the input range of the Plantard arithmetic without modifying its computation steps, and give the proof. After tailoring the Plantard arithmetic for Kyber's modulus, we show that the input range of the Plantard multiplication by a constant can be at least 2.14× larger than the work in TCHES2022 [63].

- To provide a comprehensive understanding of Plantard arithmetic, we introduce another variant of signed Plantard arithmetic proposed in a concurrent work [15]. Additionally, we present an optimization technique to eliminate one rounding operation, enhancing its efficiency on platforms that lack inherent support for right-shifting with rounding approximation. Furthermore, we offer an alternative correctness proof of this arithmetic within the CRT framework, as outlined in [113].

- Finally, we detail the advantages of the improved Plantard arithmetic over its original version, signed Plantard arithmetic in [15], Montgomery and Barrett arithmetic. Compared to the original Plantard arithmetic, the improved Plantard arithmetic not only supports signed integers but also extends its input range and narrows down its output range, thus enabling better lazy reduction strategies in NTT/INTT. Besides, it inherits the advantage of the original Plantard multiplication, i.e., saving one multiplication when one of the two operands is a constant. We further demonstrate that the improved modular arithmetic also has excellent merits over the state-of-the-art Montgomery and Barrett arithmetic, making it possible to supersede the Montgomery and

Barrett arithmetic with the improved Plantard arithmetic in LBC schemes on certain platforms.

3.2 State-of-the-art modular arithmetic

In this chapter, we divide the modular reduction into two types: mod and $\text{mod}^{\pm}$, where $c \bmod q$ obtains a result in $[0, q)$ while $c \bmod^{\pm} q$ gets a signed result in $[-\frac{q}{2}, \frac{q}{2})$. The "modular reduction" in this thesis refers to both types of modular reduction. The modular multiplication $c = a \cdot b \bmod q$ is normally divided into two parts: the multiplication $c = a \cdot b$ and the modular reduction $c \bmod q$. The modular reduction $c \bmod q$ can be done by $c - \lfloor c/q \rfloor q$, where the division operation is non-constant-time in modern computers. The non-constant-time characteristic of division prevents it from being used in the implementation of cryptographic algorithms due to security concerns. The commonly used constant time modular reduction algorithms are Montgomery reduction [86] and Barrett reduction [19]. Their basic idea is to replace the non-constant-time division with constant-time multiplications and shifts.

In current LBC schemes, the known fastest modular arithmetic is the signed version of Montgomery multiplication [104], which is given in Algorithm 13. The constant β is often set to word size 2^l so that the modulus q fits in one word. For instance, $\beta = 2^{16}$ when $q < 2^{16}$ and $\beta = 2^{32}$ when $2^{16} \leq q < 2^{32}$. The algorithm replaces the division by multiplications and shifts operations. It requires 3 multiplications.

The Barrett reduction [19] approximately computes $\lfloor c/q \rfloor$ using $\lfloor c \cdot v/(2^{\log(q)-1} \cdot \beta) \rfloor$, where v is a precomputed constant ($v = \left\lfloor 2^{\log(q)-1} \cdot \beta/q \right\rfloor$) and β satisfies $q < \beta/2$. The algorithm of the signed Barrett reduction for a single-word integer c is given in [104], as shown in Algorithm 14. The Barrett reduction requires 2 multiplications, which is the same as the Montgomery reduction.

Plantard [96] proposed a new modular multiplication (Plantard multiplication) tailored for the word size moduli and claimed that the proposed algorithm outper-

Algorithm 13 Signed Montgomery multiplication [104]

Input: Operand a, b such that $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $\beta = 2^l$ with the machine word size l, the odd modulus $q \in (0, \frac{\beta}{2})$

Output: $r \equiv ab\beta^{-1} \bmod q, r \in (-q, q)$

1: $c = c_1\beta + c_0 = a \cdot b$

2: $m = c_0 \cdot q^{-1} \bmod^{\pm} \beta$ ▷ $\bmod^{\pm}$ obtains a signed product, q^{-1} is a precomputed constant

3: $t_1 = \lfloor m \cdot q/\beta \rfloor$ ▷ shift right operation

4: $r = c_1 - t_1$

5: **return** r

Algorithm 14 Signed Barrett reduction [104]

Input: Operand c in $(-\frac{\beta}{2}, \frac{\beta}{2})$, an odd modulus q satisfying $q < \frac{\beta}{2}$, and the precomputed constant $v = \left\lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \right\rfloor$

Output: $r \equiv c \bmod q, r \in [0, \frac{3q}{2}]$

1: $t = \lfloor \frac{cv}{2^{\log(q)-1} \cdot \beta} \rfloor$ ▷ signed high product and arithmetic right shift by $\log(q) - 1$

2: $t = tq \bmod \beta$

3: $t = c - t$

4: **return** r

forms other existing solutions, including Montgomery and Barrett multiplication, in some application scenarios. The original Plantard multiplication is given in Algorithm 15. Here, we denote $X \bmod 2^{l'}$ as $[X]_{l'}$, $X >> l'$ as $[X]^{l'}$, where l' is a positive integer. The Plantard multiplication also needs 3 multiplications, which is the same as Montgomery and Barrett multiplication. But the key advantage of Plantard multiplication is that one could precompute the product of b and $q^{-1} \bmod 2^{2l}$ when operand b is a constant. This introduces an $l \times 2l$-bit multiplication $a \cdot (bq')$. If the target platform can compute the $l \times 2l$-bit multiplication in one multiplication instruction, then one multiplication could be saved for the overall modular multiplication. This advantage mainly comes from the fact that the product of two operands $c = a \cdot b$ is only used once in Plantard multiplication, while its Montgomery

and Barrett counterparts use it twice (see step 2,4 in Algorithm 13 and step 2,3 in Algorithm 14).

Algorithm 15 Original Plantard multiplication [96]

Input: Unsigned integers $a, b \in [0, q]$, $q < \frac{2^l}{\phi}, \phi = \frac{1+\sqrt{5}}{2}, q' \equiv q^{-1} \bmod 2^{2l}$, where l is the machine word size

Output: $r \equiv ab(-2^{-2l}) \bmod q$ where $r \in [0, q]$

1: $r = \left[\left([[abq']_{2l}]^l + 1\right) q\right]^l$ ▷ bq' could be precomputed when b is constant

2: **return** r

Despite this advantage, it should be noted that the original algorithm is designed for unsigned integer inputs within the range $[0, q]$. It has been well-illustrated that using a signed version of modular arithmetic in NTT/INTT will further speed up the computation. Therefore, the original Plantard multiplication could be extended to support signed arithmetic and purchase faster NTT/INTT implementations.

3.3 Improved Plantard arithmetic

This section presents the improved Plantard arithmetic, which is an improved version of the Plantard arithmetic presented in [96]. Our improvement is built by first imposing a slight restriction on the modulus based on the observation of current LBC schemes (e.g., Kyber and NTTRU); then, several tweaks are proposed to improve the Plantard arithmetic. Detailed correctness proof and advantages of the improved algorithm will be presented.

3.3.1 The proposed improved Plantard multiplication

The improved Plantard multiplication is presented in Algorithm 16. Due to the adaptation of signed numbers, we denote $(X \bmod^{\pm} 2^{l'})$ as $[X]_{l'}$, $(X >> l')$ as $[X]^{l'}$ for a positive integer l'. Here, $(X >> l')$ is equivalent to $\lfloor X/2^{l'} \rfloor$ for all signed integer numbers because the right-shift of the signed integer will always round the

Algorithm 16 Improved Plantard multiplication

Input: Two signed integers $a, b \in [-q2^{\alpha}, q2^{\alpha}], q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$

2: **return** r

result towards negative infinity. We denote q as an odd modulus, l as the minimum word size (e.g., 16, 32 or 64 bits) such that $q < 2^{l-\alpha-1}$, where α is a small integer.

The improved Plantard multiplication is based on the observation that the moduli adopted by most of the existing LBC schemes are several times smaller than the restriction $q < \frac{2^l}{\phi}$ in the original algorithm [96], e.g., 12-bit modulus 3329 in Kyber, 13-bit modulus 7681 in NTTRU, 23-bit modulus 8380417 in Dilithium. Thus, there are still margins between the original constraint and the moduli adopted in LBC schemes. Besides, all of the current state-of-the-art modular arithmetic is implemented with signed integers, which can eliminate extra additions of the multiple of q to ensure a positive output of the butterfly unit. Considering that the Plantard multiplication proposed by Thomas Plantard has the advantage of multiplying a constant (i.e., one multiplication is saved) but only supports unsigned integers, we believe that adapting signed integers for this algorithm will lead to more efficient modular arithmetic in LBC schemes.

We start by first introducing a small integer $\alpha > 0$, which narrows down the range of the modulus q from $q < \frac{2^l}{\phi}$ to $q < 2^{l-\alpha-1} < \frac{2^{l-\alpha}}{\phi}$. Based on this tweak, we can then replace the original main step $r = \left[\left([[abq']_{2l}]^{l} + 1\right) q\right]^{l}$ by $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$. In order to adapt signed inputs for this algorithm, we modify the inequality that maintains the correctness of this algorithm from $0 < q2^l - p_0 q + ab < 2^{2l}$ to $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$ under certain circumstances (see Subsection 3.3.2 for more details). We will show the correctness proof and advantages of the improved algorithm in the following two parts.

3.3.2 Proof of the improved Plantard multiplication

Theorem 3.3.1 states the correctness of Algorithm 16, and we will give the specific proof below.

Theorem 3.3.1 (Correctness)**.** *Let q be an odd modulus, l be the minimum word size (power of 2 number, e.g., 16, 32, and 64) such that $q < 2^{l-\alpha-1}$, where $\alpha > 0$, then Algorithm 16 is correct for $-q2^{\alpha} \leq a, b \leq q2^{\alpha}$.*

Proof of Theorem 3.3.1. The main step of Algorithm 16 is to compute $r = \left[\left([[abq']_{2l}]^l + 2^{\alpha}\right) q\right]^l$, namely:

$$r = \left\lfloor \frac{\left(\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^{\alpha}\right) q}{2^l} \right\rfloor.$$

1. We first check that $r \in [-\frac{q+1}{2}, \frac{q}{2})$. Since $\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor \in [-2^{l-1}, 2^{l-1} - 1]$, we have

$$\begin{aligned} \left\lfloor \frac{(-2^{l-1} + 2^{\alpha})q}{2^l} \right\rfloor \leq & r \leq \left\lfloor \frac{(2^{l-1} - 1 + 2^{\alpha})q}{2^l} \right\rfloor \\ \left\lfloor -\frac{q}{2} + \frac{q}{2^{l-\alpha}} \right\rfloor \leq & r \leq \left\lfloor \frac{q}{2} + \frac{(2^{\alpha} - 1)q}{2^l} \right\rfloor. \end{aligned}$$

   Since $\frac{q}{2^{l-\alpha}} < \frac{1}{2}$ and q is an odd number, we can get $r \geq \lfloor -\frac{q}{2} \rfloor = -\frac{q+1}{2}$ from the left-hand side of the inequation. Let's consider $\frac{q}{2} + \frac{(2^{\alpha}-1)q}{2^l}$ on the right-hand side; since $q < 2^{l-\alpha-1}$, we have

$$\frac{(2^{\alpha} - 1)q}{2^l} < \frac{q2^{\alpha}}{2^l} < \frac{2^{\alpha}2^{l-\alpha-1}}{2^l} = \frac{1}{2}.$$

   Because q is an odd number, then

$$\left\lfloor \frac{q}{2} + \frac{(2^{\alpha} - 1)q}{2^l} \right\rfloor = \left\lfloor \frac{q}{2} \right\rfloor < \frac{q}{2}.$$

   Therefore, the result r lies in $[-\frac{q+1}{2}, \frac{q}{2})$.

2. Then, we check that $r = ab(-2^{-2l}) \bmod^{\pm} q$. The proof is very similar to the one in [96], except that we introduce a small integer $\alpha > 0$ and impose a stricter constraint $q < 2^{l-\alpha-1}$ on q. Since q is an odd number, there exists $q^{-1} = q^{-1} \bmod^{\pm} 2^{2l}$ and $p \equiv abq^{-1} \bmod 2^{2l}$ so that

$$pq - ab \equiv \left(abq^{-1}\right) q - ab \bmod 2^{2l} \equiv ab - ab \bmod 2^{2l} \equiv 0 \bmod 2^{2l}.$$

Therefore, $pq - ab$ is divisible by 2^{2l}, so

$$ab\left(-2^{-2l}\right) \bmod q \equiv \frac{pq - ab}{2^{2l}}.$$

Let $p = abq^{-1} \bmod^{\pm} 2^{2l}, p_1 = \left\lfloor \frac{p}{2^l} \right\rfloor, p_0 = p - p_1 2^l$. Because the right-shift of the signed integer always rounds the result towards negative infinity, we always have $p_0 \in [0, 2^l)$. Thereafter, instead of analyzing $q2^l - p_0 q + ab$ in the original work, we slightly modify the equation to $q2^{l+\alpha} - p_0 q + ab$. The correctness of the Plantard multiplication proposed by Thomas Plantard is based on the inequality: $0 < q2^l - p_0 q + ab < 2^{2l}$. We now check that our modified equation $q2^{l+\alpha} - p_0 q + ab$ also satisfies this inequality:

$$0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l} \tag{3.3.1}$$

under the restrictions $q < 2^{l-\alpha-1}, \alpha > 0,$ and $-q2^{\alpha} \leq a, b \leq q2^{\alpha}$.

(1) When $ab \in [0, q^2 2^{2\alpha}]$ and $p_0 \in [0, 2^l)$, we have

$$q2^{l+\alpha} - p_0 q + ab \geq q2^{l+\alpha} - p_0 q > q(2^{l+\alpha} - 2^l) > 0 \text{ when } \alpha > 0, \text{ and}$$

$$\begin{aligned} q2^{l+\alpha} - p_0 q + ab &\leq q2^{l+\alpha} + ab \leq q(2^{l+\alpha} + q2^{2\alpha}) \\ &< 2^{l-\alpha-1}(2^{l+\alpha} + 2^{l+\alpha-1}) \\ &= 2^{2l-1} + 2^{2l-2} < 2^{2l}. \end{aligned}$$

Therefore, we have $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$ when $ab \geq 0$ and $\alpha > 0$.

(2) When $ab \in [-q^2 2^{2\alpha}, 0)$ and $p_0 \in [0, 2^l)$, we have

$$\begin{aligned} q2^{l+\alpha} - p_0 q + ab &> q(2^{l+\alpha} - 2^l - q2^{2\alpha}) \\ &> q(2^{l+\alpha} - 2^l - 2^{l+\alpha-1}) \\ &= q(2^{l+\alpha-1} - 2^l) \geq 0 \text{ for } \alpha \geq 1, \text{ and} \end{aligned}$$

$$q2^{l+\alpha} - p_0 q + ab < q2^{l+\alpha} < 2^{l-\alpha-1} 2^{l+\alpha} = 2^{2l-1} < 2^{2l}.$$

Therefore, we also have $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$ when $ab < 0$ and $\alpha > 0$. Overall, we obtain

$$0 < \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} < 1.$$

Combining with the fact that $pq - ab$ is divisible by 2^{2l}, we have the following equation:

$$ab\left(-2^{-2l}\right) \bmod q \equiv \frac{pq-ab}{2^{2l}} \equiv \left\lfloor \frac{pq-ab}{2^{2l}} + \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} \right\rfloor \equiv \left\lfloor \frac{qp_1 2^l + q2^{l+\alpha}}{2^{2l}} \right\rfloor$$
$$\equiv \left\lfloor \frac{q(p_1 + 2^\alpha)}{2^l} \right\rfloor \equiv \left\lfloor \frac{q\left(\left\lfloor \frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^\alpha\right)}{2^l} \right\rfloor.$$

In brief, it is equivalent to $ab(-2^{-2l}) \bmod^{\pm} q = \left[\left(\left[[abq']_{2l}\right]^l + 2^\alpha\right) q\right]^l = r$ for signed inputs. □

In summary, the improved Plantard multiplication is achieved by first imposing a stricter constraint on the modulus q. Then, the inequality that guarantees the correctness is modified into $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$. Note that the improved Plantard multiplication can be easily modified into a word size modular reduction (Plantard reduction).

3.4 Another improvement on Plantard arithmetic

In this section, the detail of the Plantard arithmetic with enlarged input range will be introduced. And this improved version is further tailored for Kyber's modulus.

3.4.1 Plantard multiplication with enlarged input range

We observe that the signed-version Plantard multiplication (see Algorithm 16) proposed in [63] accepts inputs a, b in range $[-q2^\alpha, q2^\alpha]$. By replacing $q < 2^{l-\alpha-1}$ into the range of a, b, we can see that it approximately covers most of the l-bit signed integers $(-2^{l-1}, 2^{l-1})$. Therefore, the Plantard reduction, which takes $a \cdot b$ as input, can only accept input in $(-2^{2l-2}, 2^{2l-2})$, and it cannot cover all the $2l$-bit signed integers (i.e., $[-2^{2l-1}, 2^{2l-1})$).

Through carefully theoretical analysis, we found that one can maximize the input range of the Plantard arithmetic and tailor it for a specific modulus, which

Algorithm 17 Plantard multiplication with enlarged input range

Input: Two signed integers a, b such that $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha}), q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $r = \left[\left([[abq']_{2l}]^l + 2^\alpha\right) q\right]^l$

2: **return** r

contributes to a better NTT/INTT implementation on low-end 32-bit platforms. The proposed Plantard multiplication with enlarged input range is shown in Algorithm 17. The proposed algorithm does not modify any computation steps compared with Algorithm 16, except that we manage to further enlarge the input range of ab from $[-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$ to $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. Specifically, by replacing q by $q < 2^{l-\alpha-1}$ in the above input ranges, one could get a at least two times larger input range compared to the original design.

3.4.2 Theoretical proof

Note that in 2024, Yang et al. [113] identified a misuse of the floor and ceiling functions in the correctness proof of improved Plantard arithmetic in [63]. We then updated the correctness proof to rectify this issue in the eprint version [64]. The main update we made to the algorithm was to ensure $\alpha > 0$ instead of $\alpha \geq 0$ in the original paper, which can ensure the correctness of the Algorithm 16. In this section, we show that after the update of the input range of the improved Plantard arithmetic, Algorithm 15 is actually a special case of Algorithm 17 when $\alpha = 0$. Mostly, the theoretical proof of the Plantard multiplication with enlarged input range is similar to the one given in [63] and [64]. Theorem 3.4.1 shows the correctness of Algorithm 17.

Theorem 3.4.1 (Correctness)**.** *Let q be an odd modulus, l be the minimum word length (i.e., power of 2 number) such that $q < 2^{l-\alpha-1}$, where $\alpha \geq 0$, then Algorithm 17 is correct for $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. Algorithm 15 is actually a special case of Algorithm 17 with $\alpha = 0$, supporting unsigned integers $ab \in [0, 2^{2l} - q2^l)$.*

Proof of Theorem 3.4.1. All preconditions in Algorithm 17 remain the same as in Algorithm 16 except the range of ab. We show that the input range of the Plantard multiplication can be maximized without modifying any computation steps. Similar to the proof in [63], the correctness of this algorithm depends on the following four conditions:

1. $r \in [-\frac{q+1}{2}, \frac{q}{2})$;
2. $pq - ab$ is divisible by 2^{2l}, where $p \equiv abq^{-1} \bmod 2^{2l}$;
3. $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$, where $p_1 = \left\lfloor \frac{p}{2^l} \right\rfloor, p_0 = p - p_1 2^l$. Because the right-shift of the signed integer always rounds the result towards negative infinity, we always have $p_0 \in [0, 2^l)$.
4. $r = ab(-2^{-2l}) \bmod^{\pm} q$.

Since we do not modify any preconditions and computation steps of the Plantard multiplication, the first two conditions remain true. More specifically, for condition $r \in [-\frac{q+1}{2}, \frac{q}{2})$, since $\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor$ is the only part containing ab and it is still in $[-2^{l-1}, 2^{l-1} - 1]$ even though the range of ab is increased. So, r is still in $[-\frac{q+1}{2}, \frac{q}{2})$. For condition that $pq - ab$ is divisible by 2^{2l}, because q is an odd modulus, there always exist a $p \equiv abq^{-1} \bmod 2^{2l}$ such that $pq - ab \equiv 0 \bmod 2^{2l}$. So, this condition is also valid.

The third condition is the most important part that demonstrates the correctness of the Plantard multiplication. In order to maximize its input range, we first assume that this condition holds for a new range of ab, i.e.,

$$0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}, \tag{3.4.2}$$

where $q < 2^{l-\alpha-1}, p = abq^{-1} \bmod^{\pm} 2^{2l}, p_1 = \left\lfloor \frac{p}{2^l} \right\rfloor, p_0 = p - p_1 2^l, p_0 \in [0, 2^l)$ and $\alpha \geq 0$. Then, we can deduce the maximum input range of ab through Inequality (3.4.2) as follows.

- When $ab \geq 0$ and $0 \leq p_0 < 2^l$, to ensure that the Inequality (3.4.2) holds, for the left-hand side of the inequality, we have:

$$q2^{l+\alpha} - p_0 q + ab \geq q2^{l+\alpha} - p_0 q > q2^{l+\alpha} - q2^l > 0,$$

which always holds for $\alpha \geq 0$. As for the right-hand side, we have:

$$q2^{l+\alpha} - p_0 q + ab \leq q2^{l+\alpha} + ab < 2^{2l}.$$

To ensure that the Inequality (3.4.2) holds, we conclude that $ab < 2^{2l} - q2^{l+\alpha}$.

- When $ab < 0$ and $0 \leq p_0 < 2^l$, for the right-hand side, we have:

$$q2^{l+\alpha} - p_0 q + ab < q2^{l+\alpha} < 2^{l-\alpha-1}2^{l+\alpha} < 2^{2l},$$

which always holds for $\alpha \geq 0$. As for the left-hand side, we have:

$$q2^{l+\alpha} - p_0 q + ab > q2^{l+\alpha} - q2^l + ab > 0.$$

To ensure that the Inequality (3.4.2) holds, we conclude that $ab \geq q2^l - q2^{l+\alpha}$. Therefore, the Inequality (3.4.2) is valid for every $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. By dividing every components in Inequality (3.4.2) by 2^{2l}, we can get

$$0 < \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} < 1.$$

Overall, we conclude that

$$\begin{aligned} r = ab\left(-2^{-2l}\right) \bmod q &\equiv \frac{pq - ab}{2^{2l}} \\ &\equiv \left\lfloor \frac{pq - ab}{2^{2l}} + \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} \right\rfloor \\ &\equiv \left\lfloor \frac{q\left(\frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} + 2^\alpha\right)}{2^l} \right\rfloor \end{aligned}$$

Finally, we obtain that $r = ab(-2^{-2l}) \bmod^{\pm} q = \left[\left(\left[[abq']_{2l}\right]^l + 2^\alpha\right) q\right]^l$ for signed inputs a and b that satisfy $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$.

When $\alpha = 0$, the Plantard reduction supports unsigned input $ab \in [0, 2^{2l} - q2^l)$, which is the same as the unsigned Plantard arithmetic (Algorithm 15) in [96]. This is just a special case of Algorithm 17. □

3.4.3 Plantard multiplication by a constant for Kyber

As mentioned before, Plantard arithmetic enables efficient modular multiplication by a constant in NTT/INTT, where the constant is the precomputed twiddle factor. During precomputation, one could always make sure that the twiddle factors are reduced to $[0, q)$. As the product ab lies in $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$, when we limit the constant b in $[0, q)$, the input range of a can be further enlarged.

For Kyber, we have $q = 3329$, $l = 16$. To ensure that modulus restriction $q < 2^{l-\alpha-1}$ holds, the maximum α is equal to 3. Besides, the maximum value of b (i.e. b_{max}) is equal to 3328 in Kyber. Overall, we can deduce the maximum and minimum value of a by:

$$a_{max} < (2^{2l} - q2^{l+\alpha})/b_{max}$$
$$= (2^{32} - 3329 \times 2^{19})/3328 \approx 230.13q.$$

$$a_{min} > (q2^l - q2^{l+\alpha})/b_{max}$$
$$= (3329 \times 2^{16} - 3329 \times 2^{19})/3328 \approx -137.85q.$$

Thanks to our improvements, the modular multiplication by a constant in Kyber supports input of range $a \in [-137q, 230q]$. Compared to the range $[-64q, 64q]$ in [63, Algorithm 11], the input range of the Plantard multiplication by a constant tailored for Kyber is at least 2.14× larger.

3.5 Another variant of signed Plantard arithmetic

In 2022, Aoki et al. [15] proposed a concurrent approach to improve Plantard arithmetic for signed integers, adapting it to the 32-bit NTT implementation of Saber on a 64-bit RISC-V microcontroller. Algorithm 18 illustrates the computation process of their version of signed Plantard multiplication. The primary distinction between our approach and theirs lies in the approximation methods used for the right-shifting operation in Plantard arithmetic. As discussed in Section 3.3, the right-shifting

operation in Algorithm 16 employs rounding towards negative infinity $\lfloor \cdot \rfloor$ as the approximation method, while Algorithm 18 utilizes the rounding-to-nearest $\lfloor \cdot \rceil$ approximation. Although the use of the $\lfloor \cdot \rceil$ function looks may appear to reduce the addition by 2^{α} in Algorithm 16, modern computers and microcontrollers typically implement the right-shifting operation with $\lfloor \cdot \rfloor$ approximation rather than $\lfloor \cdot \rceil$ approximation. Consequently, Aoki et al. simulated the $\lfloor \cdot \rceil$ function by adding $0x800000000$ followed by a right-shifting operation, as detailed in [15, Source code 1]. On platforms that do not natively support right-shifting with $\lfloor \cdot \rceil$ approximation, this variant of Plantard multiplication incurs one additional addition compared to our approach.

Algorithm 18 Signed Plantard multiplication [15]

Input: Two signed integers a, b with $|a|, |b| \leq 2^{l-1}$, the odd modulus $q < 2^{l-1}$ and $q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in [-\frac{q-1}{2}, \frac{q-1}{2}]$

1: $r = abq' \bmod^{\pm} 2^{2l}$

2: $r = \lfloor r/\beta \rceil$

3: $r = \lfloor rq/\beta \rceil$

4: **return** r

3.5.1 Reducing one rounding approximation

Given this limitation, Zhang et al. [116] present an improvement to reduce the $\lfloor \cdot \rceil$ approximation in Step 2 of Algorithm 18, which helps to eliminate one addition on platforms that do not natively support right-shifting with $\lfloor \cdot \rceil$ approximation.

Let $p = abq^{-1} \bmod^{\pm} 2^{2l}$. The correctness proof in [15] ensures $p_0 = p \bmod^{\pm} 2^l$ and $p_1 = \lfloor p/2^l \rceil = (p - p_0)/2^l$. Instead of using $\lfloor \cdot \rceil$, Zhang et al. [116] reused the relationship in Section 3.3, namely $p_0 = p \bmod 2^l$ and $p_1 = \lfloor p/2^l \rfloor = (p - p_0)/2^l$. The correctness of signed Plantard multiplication in [15] relies on the inequality $|ab - p_0 q| < 2^{2l-1}$. Using the above relationship in Section 3.3, the maximum value of p_0 increases from 2^{l-1} to 2^l. Consequently, the range of q should be reduced from

2^{l-1} down to 2^{l-2} to maintain the correctness. Since most LBC schemes, especially Kyber and Dilithium, satisfy $q < 2^{l-2}$, this stricter constraint on the modulus does not affect their practicability. Consequently, the remainder of the correctness proofs remains unchanged, and we refer to the detailed correctness proof in [15]. Overall, we have:

$$\frac{pq-ab}{2^{2l}} = \left\lfloor \frac{pq-ab}{2^{2l}} + \frac{ab-p_0q}{2^{2l}} \right\rceil = \left\lfloor \frac{p_1q}{2^l} \right\rceil = \left\lfloor \frac{\left\lfloor \frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor q}{2^l} \right\rceil. \tag{3.5.3}$$

In summary, this improvement replaces the $\lfloor \cdot \rceil$ function in Step 2 of Algorithm 18 with the $\lfloor \cdot \rfloor$ function, which is commonly supported by most modern CPUs. This change eliminates one addition required to simulate the $\lfloor \cdot \rceil$ while maintaining the same instruction count as Algorithm 16.

3.6 The CRT interpretation of Plantard arithmetic

In 2024, Yang et al. [113] proposed a novel interpretation of Montgomery-like modular arithmetic from the perspective of the CRT [99], offering fresh insights into the derivation of Montgomery-like modular arithmetic. Since Plantard arithmetic is a variant of Montgomery-like modular arithmetic, we also present an alternative correctness proof for Plantard arithmetic from the CRT perspective here. As for its interpretation of Montgomery arithmetic, we refer to [113] for more details.

The CRT interpretation of Montgomery-like algorithms is based on the following proposition:

Proposition 3.6.1. *Let $q, R > 1$ be two coprime integers. Denote $q^{-1} = q^{-1} \bmod R$, $R^{-1} = R^{-1} \bmod q$. Then*

$$qq^{-1} + RR^{-1} = 1 + qR. \tag{3.6.4}$$

The correctness of Theorem 3.6.1 can be readily verified by taking modulo q and modulo R on both sides of Equation 3.6.4, respectively. For a detailed theoretical proof of Theorem 3.6.1, we refer to [113, Section 3].

Proposition 3.6.2. *Algorithm 17 is correct.*

Proof of Theorem 3.6.2 in the CRT framework. Recall that the improved Plantard multiplication in Algorithm 17 computes:

$$r = \left\lfloor \frac{(\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^{\alpha}) \times q}{2^l} \right\rfloor, \tag{3.6.5}$$

for $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha}), q' = q^{-1} \bmod^{\pm} 2^{2l}, \alpha \geq 0$.

Let $R = 2^{2l}, q^{-1} = q^{-1} \bmod R, R^{-1} = R^{-1} \bmod q, c = ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. Both q^{-1} and R^{-1} are positive integers. Multiply both sides of Equation 3.6.4 by c, we get:

$$cqq^{-1} + cRR^{-1} = c + cqR.$$

Since $q' = \begin{cases} q^{-1} & \text{if } q^{-1} < \frac{R}{2} \\ q^{-1} - R & \text{if } q^{-1} \geq \frac{R}{2} \end{cases}$, we have

$$cqq' + cR^{-1}R = c + \delta cqR,$$

where $\delta = \begin{cases} 1 & \text{if } q^{-1} < \frac{R}{2} \\ 0 & \text{if } q^{-1} \geq \frac{R}{2} \end{cases}$. There exists an $\ell \in \mathbb{Z}$ such that $cq' - cq' \left(\bmod^{\pm} R\right) = \ell R$; we see that

$$\left(cq' \left(\bmod^{\pm} R\right)\right) q + cR^{-1}R = c + (\delta c - \ell)qR$$

This indicates that $\left(cq' \left(\bmod^{\pm} R\right)\right) q - c$ is divisible by R and

$$\frac{\left(cq' \left(\bmod^{\pm} R\right)\right) q - c}{R} = c\left(-R^{-1}\right) + (\delta c - \ell)q.$$

Therefore,

$$c\left(-R^{-1}\right) \left(\bmod^{\pm} q\right) = \frac{\left(cq' \left(\bmod^{\pm} R\right)\right) q - c}{R} \left(\bmod^{\pm} q\right).$$

Let $h = cq' \left(\bmod^{\pm} R\right), r' = \frac{hq-c}{R}$, the correctness of the improved Plantard arihtmetic (Algorithm 17) requires

$$\frac{hq - c}{R} = \left\lfloor \frac{\left(\left\lfloor \frac{h}{2^l} \right\rfloor + 2^{\alpha}\right) q}{2^l} \right\rfloor, \tag{3.6.6}$$

namely:

$$\frac{\left(\left\lfloor \frac{h}{2^l} \right\rfloor + 2^{\alpha}\right) q}{2^l} - 1 < \frac{hq - c}{R} \leq \frac{\left(\left\lfloor \frac{h}{2^l} \right\rfloor + 2^{\alpha}\right) q}{2^l}. \tag{3.6.7}$$

For the right-hand side of Equation 3.6.7, we have

$$\begin{aligned}\frac{hq-c}{R} &= \frac{\frac{h}{2^l}q}{2^l} - \frac{c}{2^{2l}} \\ &\leq \frac{(\lfloor \frac{h}{2^l} \rfloor + 1)q}{2^l} - \frac{c}{2^{2l}} \\ &\leq \frac{(\lfloor \frac{h}{2^l} \rfloor + 1)q}{2^l} - \frac{q2^l - q2^{l+\alpha}}{2^{2l}} \\ &= \frac{(\lfloor \frac{h}{2^l} \rfloor + 1)q + q(2^\alpha - 1)}{2^l} \\ &= \frac{(\lfloor \frac{h}{2^l} \rfloor + 2^\alpha)q}{2^l}.\end{aligned}$$

As for the left-hand side of Equation 3.6.7, we have

$$\begin{aligned}\frac{hq-c}{R} &= \frac{\frac{h}{2^l}q}{2^l} - \frac{c}{2^{2l}} \\ &> \frac{\frac{h}{2^l}q}{2^l} - \frac{2^{2l} - q2^{l+\alpha}}{2^{2l}} \\ &= \frac{\frac{h}{2^l}q + q2^\alpha}{2^l} - 1 \\ &= \frac{(\frac{h}{2^l} + 2^\alpha)q}{2^l} - 1 \\ &> \frac{(\lfloor \frac{h}{2^l} \rfloor + 2^\alpha)q}{2^l} - 1.\end{aligned}$$

Therefore, Equation 3.6.6 holds.

□

3.7 Comparisons

Original Plantard multiplication. Although the main steps between the improved and original Plantard multiplications have only a minor difference, our tweaks make it more efficient and practical in LBC schemes thanks to the following merits.

- The improved algorithm supports signed inputs and produces a result in $[-\frac{q+1}{2}, \frac{q}{2})$, while the original version only accepts unsigned integers in $[0, q]$. The advantage of using signed integers in LBC schemes has been fully discussed in previous work, i.e., eliminating an extra addition during the butterfly computation.

- Besides, our tweaks extend the input range of Plantard reduction from $[0, q^2]$ up to $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$ and change the output range from $[0, q]$ to $[-\frac{q+1}{2}, \frac{q}{2})$, thus eliminating the final correction in the original version [96, Algorithm 8]. We show that when $\alpha = 0$, the original Plantard arithmetic is actually a special case of the improved Plantard arithmetic. When $\alpha > 0$, the input range of the improved Plantard arithmetic is larger than the original version. A larger input range is especially useful in NTT/INTT because supporting more redundancy can reduce unnecessary modular reduction on the coefficients (refer to the lazy reduction strategy in Section 4.4).

In addition, compared to the signed variant of Plantard arithmetic proposed in [15], the proposed improved Plantard arithmetic exclusively utilizes the $\lfloor \cdot \rfloor$ function, which is supported by most modern CPUs. Consequently, it is more practical for implementation on such platforms. On architectures that do not natively support right-shifting with $\lfloor \cdot \rceil$ approximation, the proposed Plantard multiplication is one addition faster than the variant in [15].

Montgomery and Barrett arithmetic. The improved modular arithmetic also has several merits over the state-of-the-art Montgomery and Barrett arithmetic in LBC schemes.

- First of all, as introduced in [96], the intrinsic advantage of the Plantard multiplication is that it costs one multiplication fewer than Montgomery and Barrett multiplication when one of the two inputs is fixed. This is achieved by precomputing the product of q' and the constant input modulo 2^{2l}. Moreover, the Barrett arithmetic may require an explicit shift operation for a non-word-size offset, which will further decrease its efficiency. We refer readers to Section 4.4.1 for a detailed comparison with the Barrett arithmetic.

- Secondly, the improved Plantard reduction accepts input in $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. Compared with the Montgomery reduction that accepts $[-2^{l-1}q, 2^{l-1}q]$, the improved Plantard reduction supports more redundancy if α is big enough.

Therefore, we recommend choosing the largest α that satisfies the prerequisites so that the improved algorithm has the largest input range. The input range of the proposed modular reduction is hard to compare with the Barrett reduction directly because the input range of the Barrett arithmetic depends on the parameter setting, i.e., q and γ in Algorithm 14. The Barrett reduction used inside NTT/INTT in Kyber and NTTRU only needs to reduce 16-bit signed integers. Our modular reduction (see Algorithm 27) can also cover all 16-bit signed integers and can be a perfect replacement for the Barrett reduction in the NTT/INTT of these LBC schemes.

- The output range of the improved algorithm is narrowed down to $[-\frac{q+1}{2}, \frac{q}{2})$, which is the same as the state-of-the-art Barrett reduction presented in [3, Algorithm 3.2]. Compared with the output range $[-q, q]$ of the Montgomery multiplication, the improved Plantard multiplication halves or slows down the growing rate of the coefficient size in the NTT with CT butterflies or the INTT with GS butterflies, respectively. Together with the larger input range, the improved Plantard multiplication enables better lazy reduction strategies in NTT/INTT (see Section 4.4.2).

With all these merits over the Montgomery and Barrett arithmetic, the improved Plantard arithmetic has two weak spots. If we precompute the product of q' and the twiddle factors modulo 2^{2l} in NTT, the size of the precomputed intermediate result will be doubled. First, it needs to deal with the $l \times 2l$-bit multiplication, which may not be applicable to architectures like AVX2 and NEON. However, we will show that this method is perfectly suitable for Cortex-M4, Cortex-M7 and some 32-bit microcontrollers in Chapter 4. Second, the double-size twiddle factors are normally placed into the Read-Only Memory (ROM) instead of the Random Access Memory (RAM). Consequently, the side effect is that we need more cycles to load them into registers (see Section 4.4.2 for detailed discussion); this will not affect the stack usage. Nevertheless, arithmetic and experimental analyses in Chapter 4 show that the overall cycle reduction obtained from the aforementioned merits could offset the

cycle increment introduced by twiddle factor loading.

3.8 Conclusion

This chapter first reviewed the fundamental and state-of-the-art modular arithmetic employed in LBC schemes. It then presented the improved Plantard arithmetic, which supports modular arithmetic over signed integers in an extended range. This chapter presented correctness proofs from two distinct perspectives and includes detailed comparisons between the improved Plantard arithmetic and the original Plantard arithmetic, the signed Plantard arithmetic proposed in [15], as well as state-of-the-art Montgomery and Barrett arithmetic. These comparisons highlighted the excellent merits of the proposed Plantard arithmetic, demonstrating its potential to replace existing modular arithmetic methods in LBC schemes on specific platforms.

Chapter 4

Efficient Lattice-based Cryptography on IoT Devices

This chapter includes the optimized LBC implementations on ARM Cortex-M4, ARM Cortex-M3, and RISC-V, which are presented in our three previous works [63, 65, 62].

4.1 Introduction

Chapter 3 introduced the improved Plantard arithmetic, demonstrating that its better efficiency, bigger input range and smaller output range compared to the original Plantard, state-of-the-art Montgomery, and Barrett arithmetic. However, as discussed in Section 3.7, the improved Plantard arithmetic also has some limitations that could affect its feasibility and practicality on certain target platforms. Despite the mathematical improvements, deploying this arithmetic on these platforms requires further exploration to address these shortcomings from both electrical engineering and computer science perspectives. This underscores that cryptographic engineering is an interdisciplinary field. To effectively deploy the improved Plantard arithmetic and leverage it for accelerating the LBC schemes on IoT devices, a thorough analysis of both the computational steps involved in the arithmetic and the ISA characteristics of the target platforms is crucial. This analysis will reveal

whether the platform's instruction set is well-suited for the main computations in the Plantard arithmetic. By aligning the computational steps of the Plantard arithmetic with the ISA features of the target platform, we can evaluate the feasibility of its efficient implementation and assess whether it can deliver performance improvements for the NTT/INTT operations in LBC schemes, particularly in IoT devices with very fundamental instruction set.

In addition, the extensive research on polynomial arithmetic [104, 32, 12, 3, 57, 2, 23] has already achieved excellent results in LBC schemes. Recent research [57] shows that the polynomial multiplication of Dilithium accounts for mainly less than 10% of the running time on Cortex-M4. According to PQM4 [71], the most time-consuming operation in LBC schemes is actually Keccak, which is a multipurpose cryptographic primitive notably known for being the core of the SHA-3 and SHAKE standards [45]. Keccak is extensively used by many PQC schemes for various purposes, from seed expansion to CPA-to-CCA transforms. For example, Dilithium and Kyber, another LBC scheme to be standardized as ML-KEM [90], rely on Keccak to such an extent that hashing accounts for 85% of the running time on Cortex-M4 [57]. While Keccak runs fast on high-end processors thanks to vectorization or dedicated instructions (e.g. ARMv8 SHA-3 extension) [24], its largest version does not show outstanding performance on constrained platforms (e.g. ARMv7-M microcontrollers) when fully implemented in software. This is mainly due to the lack of general-purpose registers to hold the entire 1600-bit internal state, leading to high register pressure and therefore register spilling (i.e. saving and restoring some intermediate variables to and from memory). By way of illustration, on ARM Cortex-M4 processors, the current fastest Keccak implementation spends around half of the running time in memory accesses. To reduce this overload, the Keccak designers suggested using a 12-round variant named TurboSHAKE [30] instead of the 24-round variant, but NIST was not in favour of such a decision[1]. This highlights the need for improvement on architectures such as ARMv7-M where Keccak does

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/5HveEPBsbxY

not show outstanding performance.

Contributions. Therefore, in order to fully enhance the performance of LBC schemes, this chapter not only investigates the optimizations of the proposed Plantard arithmetic and polynomial multiplication but also presents dedicated optimizations for Keccak on 32-bit IoT platforms. Detailed contributions of this chapter can be summarized as follows:

- We present an efficient implementation of the improved Plantard arithmetic for 16-bit odd moduli using the `smulw{b,t}` instruction to perform the 16×32-bit multiplication on Cortex-M4. As for low-end 32-bit platforms, we propose two optimization techniques to implement the 16-bit Plantard arithmetic utilizing the specific ISA characteristics of Cortex-M3 and RISC-V. The optimized implementations demonstrate that the Plantard arithmetic can indeed supersede the state-of-the-art Montgomery and Barrett arithmetic on these low-end 32-bit microprocessors. Furthermore, we show that the Plantard arithmetic can be easily extended to other schemes or platforms with similar ISA characteristics, which highlights its extensibility.

- For the 16-bit polynomial multiplication in Kyber and NTTRU, by replacing the state-of-the-art Montgomery and Barrett arithmetic with the proposed Plantard arithmetic, we demonstrate that it enables more efficient polynomial arithmetic, namely NTT, INTT, and modular reduction, leading to faster Kyber and NTTRU implementations on three 32-bit platforms: Cortex-M4, Cortex-M3 and RISC-V. The shortcomings of the Plantard arithmetic can be easily compensated for by the performance gains it offers. More specifically, the efficient Plantard multiplication by a constant can significantly speed up the modular multiplication by a twiddle factor in the butterfly unit, using proper precomputation, and thus requires one fewer multiplication compared to Montgomery arithmetic. Moreover, with the larger input range and smaller output range of the Plantard arithmetic compared to state-of-the-art Montgomery

arithmetic, it enables a more efficient lazy reduction strategy, reducing or entirely eliminating the need for modular reduction of coefficients in NTT/INTT. Our NTT and INTT implementations show significant speedups compared to the state-of-the-art, achieving speedups of 25.02%/20.84%, 26.19%/32.57%, 34.76%/55.53%, and 22.67%/43.37% on Cortex-M4, Cortex-M3, SiFive RISC-V board, and PQRISCV, respectively.

- For the polynomial multiplication in Dilithium, which typically uses 32-bit NTT, we revisit the idea of using small NTT (i.e., 16-bit NTT) for $c\mathbf{s}_i$ and multi-moduli NTT for $c\mathbf{t}_i$ on ARMv7-M. While the potential benefits of integrating the multi-moduli NTT into Dilithium have been briefly discussed in [57], the authors claimed that the multi-moduli NTT implementation using Montgomery arithmetic could not outperform their constant-time 32-bit NTT implementation on Cortex-M3. We show that by leveraging the efficient Plantard arithmetic, the proposed multi-moduli NTT implementation on Cortex-M3 remains constant-time and can indeed yield over 19.07% or 52.76% speed-ups when compared with the variable-time or constant-time 32-bit NTT in [57], respectively. Specifically, we construct the multi-moduli NTT using NTTs over 769 and 3329. By utilizing the excellent properties of Plantard arithmetic, we eliminate all modular reductions in NTT and INTT over 769 and 3329. Moreover, we also show that the efficient Plantard arithmetic can be used to accelerate the explicit CRT by 19.45% in the multi-moduli NTT implementation, further improving the feasibility and efficiency of the multi-moduli NTT on Cortex-M3.

- In addition, we provide two versions of Kyber implementations, namely the stack-friendly (stack-version) and high-speed (speed-version) implementations on Cortex-M4, Cortex-M3 and RISC-V, in line with [3]. To deploy the speed-version Kyber implementation on memory-constrained IoT devices, we propose two memory optimization strategies that lead to a significant reduction in stack usage, ranging between 23.50% to 28.31% compared to the stack us-

age in [3]. This makes it more feasible for deployment on memory-constrained IoT devices. Overall, our optimized Kyber implementations achieve new speed records on both Cortex-M3 and RISC-V. Specifically, the speed-version outperforms PQM3 by a margin of 3.69%∼5.63%, 3.51%∼5.15%, and 3.37%∼4.67% for Kyber512, Kyber768, and Kyber1024, respectively, on Cortex-M3. On PQRISCV, it outperforms previous work by 13.59%∼27.03%, 25.49%∼31.15%, and 26.96%∼31.43% using only 26.86%∼52.44% of their stack usage for the three Kyber variants, respectively.

- Finally, we improve Keccak's performance on Cortex-M3 and M4 by up to 24.78% and 21.4% utilizing two architecture-specific optimizations. The first one, denoted as lazy rotations, consists of taking advantage of the inline barrel shifter to eliminate explicit rotations in the linear layer. Note that this technique has been reported in the literature by Becker and Kannwischer in order to boost Keccak on ARMv8 architectures [24] but has not been ported to ARMv7-M so far. The second optimization consists of a more efficient memory access scheduling to avoid pipeline hazards, which results in over 12.84% performance improvements on ARMv7-M.

4.2 Target platforms

This thesis targets the following IoT platforms: Cortex-M4, Cortex-M3, and RISC-V. The Cortex-M4 is one of the IoT platforms recommended by NIST for evaluating the performance of PQC schemes. The Cortex-M3, with its relatively limited instruction set, requires special consideration in cryptographic engineering. RISC-V, as a widely adopted open-source instruction set architecture, has gained significant popularity in recent years. These platforms are highly relevant to IoT scenarios and are therefore selected as the focus of this thesis.

4.2.1 ARM Cortex-M4

The ARM Cortex-M4 processor is a part of the ARMv7-M microcontroller profile, featuring sixteen 32-bit registers (`r0-r15`). Among these, `r13` is reserved as the stack pointer, `r14` as the link register, and `r15` as the program counter. Because Cortex-M4 supports floating-point (FP) computation, 32 32-bit FP registers are available for programming and can be used to "cache" frequently-used values.

The Cortex-M4 employs a three-stage pipeline, where most instructions execute in a single cycle. Exceptions include branches, multiplications, and memory operations, which may require additional cycles. For instance, `ldr` and `str` instructions typically take 2 and 1 cycles, respectively, though dependencies can increase `ldr` latency to up to 3 cycles. When no dependencies exist, multiple `ldr` instructions can be pipelined, executing in $n + 1$ cycles for n operations, while a `str` following an `ldr` incurs no additional delay. The Cortex-M4 also supports Single Instruction Multiple Data (SIMD) instructions that can operate two halfwords or four bytes in parallel. The essential instructions in our implementation include `smul{b,t}{b,t}`, `smulw{b,t}`, and `ldrd`. We use Rd to represent the destination register; Rm and Rn represent two source registers. The `smulbb` instruction multiplies the bottom halfword of Rm by the bottom halfword of Rn and writes the result to Rd. The `smulwb` instruction multiplies Rm by the bottom halfword of Rn; extracts the most significant 32 bits of the result, and writes it to Rd. The `ldrd` R8, R9, [R3, #0x20] instruction loads R8 from the address R3+32 and loads R9 from the address R3+36. Equipped with a barrel shifter, the Cortex-M4 allows operand shifts or rotations within instructions without performance penalties, except when the shift amount is specified by a register, which introduces a single-cycle overhead. Most multiplication instructions execute in a single cycle.

4.2.2 ARM Cortex-M3

The ARM Cortex-M3 platform we use is Arduino Due which integrates an Atmel SAM3X8E core [44]. It comprises 96 KiB of RAM and 512 KiB of Flash, and oper-

ates at a maximum frequency of 16 MHz. Similar to Cortex-M4, the Cortex-M3 platform has 16 32-bit general-purpose registers, of which fourteen are programmable. However, it doesn't provide any floating-point (FP) registers and SIMD extensions. It should be noted that there are some non-constant-time instructions like `umull, smull, umlal` and `smlal` [40] that we need to avoid using during the cryptographic implementation, otherwise it may suffer from timing attacks. The only two "secure" multiplication instructions one can use in this platform are the `mul` and `mla` instructions for computing the low 32 bits of the 64-bit product. The `mul` instruction costs one cycle while the `mla` instruction takes two cycles. Most of the other instructions are 1-cycle, while the memory access operations such as `ldrh, ldr, ldrd, strh, str` and `strd` are relatively expensive. Loading or storing half-word/word requires two cycles, and loading or storing double-word (`ldrd/strd`) takes three cycles. Cortex-M3 also provides a useful inline barrel shifter operation, similar to Cortex-M4, which performs an additional shift operation before the addition and subtraction instructions, without additional overhead. For instance, the instruction (`add.w rd, rn, rm, asr #16`) first right-shifts the operand `rm` by 16 bits, and subsequently, the addition is performed using the right-shifted result.

4.2.3 RISC-V

RISC-V is an open-source standard Instruction Set Architecture (ISA), enabling a new era of processor innovation through open collaboration. Because of its open-source features, it has attracted huge attention from both research community and industry. We select two RISC-V platforms as the benchmark platforms. The first one is the SiFive Freedom E310 board, which integrates a 32-bit E31 RISC-V core [106]. The RV32IMAC instruction set is instantiated in this core. This platform is a memory-constrained IoT device with only 16KiB of RAM, making it a suitable platform for measuring the performance and applicability of LBC schemes on low-end IoT devices. The second platform is a RISC-V simulator based on VexRiscv[2],

[2] `https://github.com/SpinalHDL/VexRiscv`

as described in pqriscv-vexriscv[3] and used in PQRISC[4]. The RV32IM instruction set is instantiated in this simulator. In the following sections, we will denote the VexRiscv RISC-V simulator as PQRISCV. Both platforms comprise of 32 32-bit general-purpose registers, of which thirty are programmable. Similar to the Cortex-M3 platform, they do not have powerful SIMD extensions. The cycle counts of some instructions of PQRISCV are not well-specified in the documents, so we will not discuss them in detail. The cycle counts mentioned below only refer to the SiFive board. There are only two multiplication instructions (`mul, mulh`) that can be used, and both of them take five cycles on the SiFive board. Loading or storing one word (`lw, sw`) requires two cycles while loading or storing one half-word or byte (`lh, lhu, lb, lu`) takes three cycles. Apart from the multiplication, division, and memory access operations, most of the other instructions consume one cycle. All instructions are constant-time except the division.

4.3 Optimized modular arithmetic implementations

In this section, we present optimized implementations of the improved Plantard arithmetic for 16-bit word-size moduli on Cortex-M4, Cortex-M3 and RISC-V.

4.3.1 Faster 16-bit Plantard arithmetic on Cortex-M4

Before moving forward into the implementation details, it is assumed that there exists a small integer $\alpha > 0$ such that the prerequisite ($q < 2^{l-\alpha-1}$) mentioned in Theorem 3.3.1 is established (which is the case for most of the LBC schemes, including Kyber and NTTRU). The word size in the following parts is set to $l = 16$ to support the 16-bit arithmetic in Kyber and NTTRU. As recommended in Section 3.7, the maximum α that satisfies Theorem 3.3.1 is $\alpha = 3/\alpha = 2$ for Kyber/NTTRU.

[3] `https://github.com/mupq/pqriscv-vexriscv`

[4] `https://github.com/mupq/pqriscv`

Plantard multiplication by a constant. Based on the Plantard multiplication described in Algorithm 16, we propose a 2-cycle modular multiplication by a constant for 16-bit moduli. The detailed instruction sequence is shown in Algorithm 19. This is achieved by observing that a part of the main step of the Plantard multiplication is $abq' \bmod^{\pm} 2^{2l}$. If the multiplicand b is a constant, one can directly precompute $bq' \bmod^{\pm} 2^{2l}$ where $q' = q^{-1} \bmod^{\pm} 2^{2l}$. Recall that the input of Algorithm 16, a and b, satisfies $a, b \in [-q2^{\alpha}, q2^{\alpha}]$ and $ab \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$. We can always reduce the constant b down to $[0, q)$ during the precomputation of bq'. Therefore, the input range of a can be extended to $[-q2^{2\alpha}, q2^{2\alpha}]$. Besides, for the parameter α, we have $2^{l-\alpha-2} < q < 2^{l-\alpha-1}$ when the maximum α is chosen. Therefore, by combining $2^{l-\alpha-2} < q$ and $a \in [-2^{2\alpha}q, 2^{2\alpha}q]$, we conclude that a lies in $[-2^{l+\alpha-2}, 2^{l+\alpha-2}]$. For any integer $\alpha > 0$, the input a covers every 16-bit signed integer, i.e., $a \in [-2^{l-1}, 2^{l-1})$.

The main step of Algorithm 16 is $[([[abq']_{2l}]^l + 2^{\alpha})q]^l$. The `smulwb` instruction in Algorithm 19 computes $[[abq']_{2l}]^l$ in 1 cycle, and the intermediate result is saved in the bottom half of r. Here, adding 2^{α} to the bottom half of r before the multiplication with q might produce a carry to the top half of the 32-bit register r. This would cause an error when the top half of r is not zero. This problem can be addressed by computing the main step separately as $[q \cdot [[abq']_{2l}]^l + q2^{\alpha}]^l$ with the `smlabb` instruction. Here, the term $q2^{\alpha}$ is fixed and can be precomputed without extra runtime overhead. The final result locates in the top half of r. Compared with Algorithm 20, the state-of-the-art 16-bit Montgomery multiplication implementation on Cortex-M4 [12], we reduce one multiplication by precomputing the product of q^{-1} and the constant operand b. As for its comparison with the Barrett arithmetic, we refer to Section 4.4.1 for more details.

Note that [82] proposed a similar trick for Montgomery multiplication with the AVX2 instructions, which is improved over the work in [104]. It is achieved by precomputing $q^{-1}/-q^{-1}$ with the bottom half of c (see step 2 of Algorithm 13/Algorithm 20). Unfortunately, it is not suitable for Cortex-M4 because the trick shown in [82] highly

Algorithm 19 The 2-cycle improved Plantard multiplication by a constant on Cortex-M4

Input: An l-bit signed integer $a \in [-2^{l-1}, 2^{l-1})$, a precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r_{top} = ab(-2^{-2l}) \bmod^{\pm} q, r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `smulwb` r, bq', a ▷ $r \leftarrow [[abq']_{2l}]^l$

3: `smlabb` $r, r, q, q2^{\alpha}$ ▷ $r_{\text{top}} \leftarrow [q[r]_l + q2^{\alpha}]^l$

4: **return** r_{top}

Algorithm 20 The 3-cycle signed Montgomery multiplication on Cortex-M4 [12]

Input: Two l-bit signed integers a, b such that $ab \in [-q2^{l-1}, q2^{l-1})$

Output: $r_{top} = ab2^{-l} \bmod^{\pm} q, r_{top} \in (-q, q)$

1: `mul` c, a, b

2: `smulbb` $r, c, -q^{-1}$ ▷ $r \leftarrow [c]_l \cdot (-q^{-1})$

3: `smlabb` r, r, q, c ▷ $r_{\text{top}} \leftarrow [[r]_l \cdot q]^l + [c]^l$

4: **return** r_{top}

relies on the two-instruction-fashion multiplication, whereas the 16/32-bit multiplication on Cortex-M4 is carried out in a single instruction. Applying their optimization on Cortex-M4 introduces extra multiplication instructions as stated in [32, Subsection 3.2].

Plantard reduction. As for the multiplication of two variables, we present a 2-cycle Plantard reduction over a 32-bit signed product $c \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$, which is shown in Algorithm 21. Instead of using `smulwb` as we did in Algorithm 19, the `mul` instruction is used to compute $[[cq']_{2l}]^l$, and the intermediate result locates in the top half of r. The result is then obtained by the `smlatb` instruction. It is worth noting that the input range of Algorithm 21 is $q2^{2\alpha-l+1}$ times larger than the Montgomery reduction (the reduction version of Algorithm 20). Since α is chosen to ensure $q < 2^{l-\alpha-1}$, we have $q2^{\alpha+1} < 2^{l-\alpha-1} \cdot 2^{\alpha+1} = 2^l$. As for the maximum α,

Algorithm 21 The 2-cycle improved Plantard reduction on Cortex-M4

Input: A $2l$-bit signed integer $c \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$

Output: $r_{top} = c(-2^{-2l}) \bmod^{\pm} q, r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $q' \leftarrow q^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `mul` r, c, q'

3: `smlatb` $r, r, q, q2^{\alpha}$

4: **return** r_{top}

we have $q2^{\alpha+1} \approx 2^l$, then $q2^{2\alpha+1-l} = q2^{\alpha+1-l} \cdot 2^{\alpha} \approx 2^{\alpha}$. Therefore, the input range of the Plantard reduction is approximately 2^{α} times larger than the Montgomery reduction. The excellent input range and output range of the improved Plantard arithmetic enable better lazy reduction strategies in NTT/INTT (see Section 4.4.2).

4.3.2 Faster 16-bit Plantard arithmetic on Cortex-M3 and RISC-V

The efficient Plantard arithmetic on Cortex-M4 in Subsection 4.3.1 was dependent on the SIMD instruction `smulw{b,t}` for carrying out the 16×32-bit multiplication $[a \times bq']_{2l}$. Here, a denotes a 16-bit signed integer and bq' represents a precomputed 32-bit positive integer. In [63, Algorithm 16], we also presented a 5-instruction implementation of the Plantard multiplication by a constant on RISC-V, which concludes that in cases where multiplication instruction is slower than the shift instruction, the Plantard arithmetic has better performance than the Montgomery and Barrett arithmetic. Moreover, due to the use of SIMD extension on Cortex-M4, the coefficients in NTT/INTT were subjected to 16-bit signed integer in Cortex-M4 implementation, which did not fully utilized the large input range of Plantard arithmetic. The implementation on low-end 32-bit platforms stores each coefficient in a 32-bit register. It enables us to temporarily overflow 16-bit signed integer and fully utilize the large input range of the Plantard arithmetic. Therefore, its extensibility on low-end 32-bit platforms without such powerful SIMD extension

still needs more exploration.

In this section, we propose two optimization techniques to improve the performance of the Plantard arithmetic based on the specific ISA characteristics of Cortex-M3 and RISC-V, respectively. These optimization techniques show that the Plantard arithmetic can outperform the Montgomery and Barrett arithmetic without the prerequisite of slow multiplication. This is demonstrated by the efficient implementation of Plantard arithmetic on Cortex-M3, where both multiplication and shift instructions take one cycle. Notably, the efficient implementation of Plantard arithmetic is not restricted to Kyber's modulus and can be customized for other 16-bit odd moduli as well.

Plantard multiplication by a constant The first optimization technique is proposed for efficient Plantard multiplication by a constant on Cortex-M3 (Algorithm 22). Unlike the Plantard multiplication by a constant on Cortex-M4 [63, Algorithm 11], this algorithm accepts a 32-bit signed integer as input on low-end 32-bit platforms. If one implements the multiplication $[a \times bq']_{2l}$ with the 32×32-bit multiplication instruction `mul`, the valid product locates in the most significant half of the 32-bit register r. Then, one need to shift the valid product to the least positive half so as to compute the later steps. The above shift operation might decrease the efficiency of Plantard arithmetic, as it consumes an extra cycle. In order to improve the efficiency, we propose to merge this shift operation with the "add to 2^{α}" step by utilizing the inline barrel shifter operation on Cortex-M3 (cf. Step 3 of Algorithm 22). This operation initiates by right-shifting register r by 16 bits. Thereafter, the addition of 2^{α} is directly carried out in the same cycle. After that, the remaining operations are computed by using just one multiplication and one right-shift instructions. In Section 4.4.2, a similar technique will be proposed to omit the final shift operation in the CT algorithm on Cortex-M3.

The second optimization technique for the Plantard multiplication by a constant is carried out on RISC-V, as demonstrated in Algorithm 23. Similar to the algorithm on Cortex-M3, we also utilize the 32×32-bit multiplication instruction `mul` to

Algorithm 22 Efficient Plantard multiplication by a constant for Kyber on Cortex-M3

Input: An 32-bit signed integer $a \in [-137q, 230q]$, a precomputed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{32}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

1: $bq' \leftarrow bq^{-1} \bmod 2^{2l}$ ▷ precomputed

2: `mul` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$

3: `add` $r, 2^{\alpha}, r, \text{asr}\#16$ ▷ $r \leftarrow ([r]^{l} + 2^{\alpha})$

4: `mul` r, r, q

5: `asr` $r, r, \#16$ ▷ $r \leftarrow [rq]^{l}$

6: **return** r

implement the multiplication $[a \times bq']_{2l}$. As RISC-V does not provide inline barrel shifter operation as in Cortex-M3, we perform the right-shift operation and the addition of 2^{α} separately. Then, instead of computing $[rq]^{l}$ with one multiplication and one shift similar to the final two steps in Algorithm 22, we propose to precompute $q2^{l} = q \times 2^{l}$, which is inspired by the similar technique of Montgomery arithmetic proposed in [56]. This technique utilizes the fact that $[rq]^{l}$ is equivalent to $[rq2^{l}]^{2l}$, and the division by 2^{2l} operation can be implemented by using one `mulh` instruction. Note that this optimization technique cannot be applied to Cortex-M3 as it does not offer constant-time full multiplication instruction, but it can be securely extended to other platforms as long as the targeted platform has constant-time `mulh` instruction.

For platforms that lack of inline shifter and `mulh` instructions, if there is an efficient and constant-time `mla` instruction to handle the multiply-and-add operation, it is also possible to merge step 4 and 5 in [63, Algorithm 16]. But we choose not to do it on Cortex-M3 since the `mla` instruction use two cycles on this platform, which will make this version slower than that in Algorithm 22.

In summary, the efficient Plantard multiplication by a constant on both Cortex-M3 and RISC-V take four instructions, including two multiplications, one shift and one addition. Compared to the existing Plantard and Montgomery multiplication

Algorithm 23 Efficient Plantard multiplication by a constant for Kyber on RISC-V

Input: An 32-bit signed integer $a \in [-137q, 230q]$, a precomputed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{32}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `mul` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$

3: `srai` $r, r, \#16$

4: `addi` $r, r, 2^{\alpha}$ ▷ $r \leftarrow ([r]^l + 2^{\alpha})$

5: `mulh` $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$

6: **return** r

on RISC-V or Cortex-M3 [63, 56] and [46] that take five instructions, the proposed implementation is one multiplication fewer.

Plantard reduction We propose a 4-instruction Plantard reduction for the modular multiplication of two variables on both architectures, which is equally efficient as the state-of-the-art Montgomery or Barrett reduction in [56] and [46]. The proposed instruction sequence of the Plantard reduction is similar to Algorithm 22 and Algorithm 23, except that the 32-bit precomputed constant bq' is replaced by $q^{-1} \bmod 2^{2l}$ in the modular multiplication of two variables or $(-2^{2l} \bmod q) \times q^{-1} \bmod 2^{2l}$ in the modular reduction of coefficients. This variant of modular multiplication is used in the base multiplication of Kyber. Besides, previous implementations mostly utilize the Barrett reduction to implement the modular reduction of coefficients. Nevertheless, despite the same cycle consumption as the Barrett reduction on Cortex-M3 and RISC-V, we chose to adopt the Plantard reduction for the sake of consistency.

4.3.3 Plantard arithmetic on other schemes and platforms

In the previous sections, we focused on the efficient 16-bit Plantard arithmetic implementations on 32-bit platforms. Here, we demonstrate that the Plantard arithmetic is not limited to 16-bit operations on 32-bit platforms but can also be extended to

Algorithm 24 Efficient Plantard multiplication by a constant for Dilithium on RV64IM

Input: An 64-bit signed integer $a \in [-130686q, 131457q]$, a precomputed 64-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{64}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `mul` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$

3: `srai` $r, r, \#32$

4: `addi` $r, r, \#256$ ▷ $r \leftarrow ([r]^l + 2^\alpha)$

5: `mulh` $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$

6: **return** r

larger word sizes or other platforms, provided the necessary instructions are available to handle the $l \times 2l$-bit multiplication in Plantard arithmetic. This extensibility is first demonstrated through the efficient 32-bit Plantard arithmetic implementations on a 64-bit RISC-V platform, as shown in [15]. However, it is important to note that the signed Plantard arithmetic proposed by Aoki et al. requires two $\lfloor \cdot \rceil$ functions, which were implemented by simulating the rounding approximation with two extra additions, as described in [15, Source Code 1]. As a result, their implementation consumes one additional addition compared to the proposed improved Plantard arithmetic.

In several follow-up works [116, 70, 114] that I participated in, we further extended the application of the improved Plantard arithmetic to other LBC schemes and platforms. Specifically, to speed up the 32-bit modular arithmetic of Dilithium on 64-bit ISAs such as RV64IM, we can tailor the Plantard arithmetic to the Dilithium parameters and implement the 32-bit Plantard arithmetic using the same instruction sequence outlined in Algorithm 23. More specifically, as shown in Algorithm 24, the efficient Plantard multiplication by a constant for Dilithium on RV64IM requires adjusting $l = 32$ and $\alpha = 8$ for Dilithium's 23-bit moduli $q = 8380417$ [116, 70].

Moreover, we also explore the potential of customizing the RISC-V instruction

Algorithm 25 Efficient Plantard multiplication by a constant for NTRU and Hawk on customized RISC-V SIMD ISA

Input: An 32-bit signed integer a, a precomputed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{32}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `mulv` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$

3: `asravi` $r, r, 2^{\alpha}, l$ ▷ $r \leftarrow ([r]^l + 2^{\alpha})$

4: `mulvh` $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$

5: **return** r

set for the proposed Plantard arithmetic to ensure its efficient deployment on a customized 32-bit SIMD architecture [114], where each lane of the SIMD register is 32 bits. Specifically, we focus on the 16-bit modulus $q = 12289$ used in NTRU and Hawk on 32-bit SIMD architectures. Here, we simply need to adjust $l = 16$ and $\alpha = 1$ to suit the modulus $q = 12289$. The instruction sequence is similar to that in Algorithm 23, with the main difference being that the multiplication instructions `mulv` and `mulvh` are the SIMD versions of `mul` and `mulh`, respectively. Additionally, we propose a novel instruction `asravi` in Step 2 of Algorithm 25 to further accelerate the Plantard arithmetic. This instruction combines one addition and one right shift operation into a single instruction. With this optimization, the 16-bit Plantard arithmetic becomes two instructions faster than the signed Montgomery multiplication algorithm, as demonstrated in [116, Algorithm 4].

In summary, these examples demonstrate the extensibility of the proposed Plantard arithmetic to other LBC schemes across various platforms. We anticipate further exploration of its deployment on additional schemes and platforms in the future.

4.4 Kyber and NTTRU optimizations on ARMv7-M and RISC-V

This section presents the proposed performance and memory optimizations for 16-bit polynomial multiplication in NTTRU and Kyber on Cortex-M4, Cortex-M3 and RISC-V.

4.4.1 Optimized 16-bit NTT/INTT implementation on Cortex-M4

This subsection details the improvements the improved Plantard arithmetic brings to the butterfly unit, layer merging, and lazy reduction in NTT/INTT on Cortex-M4.

Butterfly unit. The core operations in NTT/INTT are performed in the butterfly unit. Commonly, there are two types of butterfly structures, namely the CT butterfly and GS butterfly. In Kyber, using CT butterfly in NTT and GS butterfly in INTT is a common strategy to avoid coefficients flipping. However, recent reports [36, 3] state that using CT butterfly for both NTT and INTT in LBC schemes would also result in faster code. For both strategies in 16-bit NTT/INTT, one needs to compute CT/GS butterfly over two 32-bit packed integers and return two 32-bit packed results [32, 12]. The improved Plantard multiplication by a constant (Algorithm 19) can be well adapted into the double CT/GS butterflies with the help of the `smulw{b,t}` instruction on Cortex-M4.

The key operation in the butterfly unit is the modular multiplication by a proper twiddle factor ζ, which is a precomputed constant. It is common to store the twiddle factors in the Montgomery domain when using Montgomery arithmetic. Similar to Montgomery arithmetic, Plantard arithmetic also produces a result in a special domain, namely $c(-2^{-2l}) \bmod^{\pm} q$. Therefore, to integrate Plantard arithmetic into the butterfly unit, we also store the twiddle factors in the "Plantard" domain by

Algorithm 26 Double CT butterfly on Cortex-M4

Input: Two 32-bit packed signed integers a, b (each containing a pair of 16-bit signed coefficients), the 32-bit twiddle factor ζ

Output: $a = (a_{top} + b_{top}\zeta)||(a_{bottom} + b_{bottom}\zeta), b = (a_{top} - b_{top}\zeta)||(a_{bottom} - b_{bottom}\zeta)$

1: `smulwb` t, ζ, b

2: `smulwt` b, ζ, b

3: `smlabb` $t, t, q, q2^{\alpha}$

4: `smlabb` $b, b, q, q2^{\alpha}$

5: `pkhtb` $t, b, t,$ asr#16

6: `usub16` b, a, t

7: `uadd16` a, a, t

8: **return** a, b

multiplying each twiddle factor with a constant $-2^{2l} \bmod q$. Besides, to utilize the improved Plantard multiplication by a constant (Algorithm 19) in the butterfly unit, one also needs to multiply $q^{-1} \bmod^{\pm} 2^{2l}$ by the twiddle factor ζ in the "Plantard" domain modulo 2^{2l}, namely $\zeta = (\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1} \bmod^{\pm} 2^{2l}$. Note that this precomputation would result in a $2l$-bit twiddle factor. Although it would introduce extra cycles for data loading, this overhead can be offset by the improvements brought by the improved Plantard arithmetic.

Algorithm 26 illustrates the detailed instruction sequence of the double CT butterfly, which computes $a = (a_{top}+b_{top}\zeta)||(a_{bottom}+b_{bottom}\zeta), b = (a_{top}-b_{top}\zeta)||(a_{bottom}-b_{bottom}\zeta)$ over packed arguments a, b. Thanks to the SIMD instruction `smulw{b,t}`, one can pack two 16-bit coefficients into one 32-bit register b. Then, one can perform the Plantard multiplication (Algorithm 19) by a proper twiddle factor ζ over the top and bottom half of b using the `smulwt` and `smulwb` instructions, respectively. The follow-up instruction sequence is the same as the previous work [12]. In summary, we obtain a 7-instruction double CT butterfly for packed arguments, which reduces 2 instructions compared with the one that uses Montgomery multiplication. Similarly, we can also obtain a 7-instruction double GS butterfly for INTT.

Layer merging. As stated in Section 4.4.2, using the improved Plantard arithmetic will double the size of the twiddle factors. It would bring extra `load` instructions to load the 32-bit twiddle factors compared with the original 16-bit version. However, the improvements brought by the improved Plantard arithmetic can easily bury this overhead thanks to the layer merging techniques. There are two efficient layer merging strategies for Kyber on Cortex-M4, which are presented in [12] and [3], respectively. The layer merging strategy in [12] is the 3-layer merging strategy (3-3-1), while [3] adopts the 4-layer merging strategy (4-3). Since the first 4-layer NTT re-uses the same 15 twiddle factors multiple times, [3] proposes to cache the 15 16-bit twiddle factors into 8 FP registers and replace the memory access instruction with the cheaper `vmov` instruction. Apart from using the FP registers and the `vmov` instruction, the 4-layer merging strategy is actually built upon the 3-layer merging strategy in [12].

The 3-layer merging strategy in [12] allows us to compute 8 butterflies at the cost of loading 1, 2, or 4 32-bit twiddle factor(s) in each layer. Note that loading 2 consecutive 32-bit twiddle factors can be achieved by a 3-cycle `ldrd` instruction on Cortex-M4, which is merely 1-cycle slower than the original `ldr` instruction to load 2 consecutive 16-bit twiddle factors. We rearrange the twiddle factors in the same order as they are used so that one can use `ldrd` to load 2 consecutive 32-bit twiddle factors. Overall, by integrating the Plantard arithmetic into the 3-layer merging strategy, one could reduce 8 cycles (reduce 1 cycle for each butterfly) with 0, 1, or 2 extra cycle(s) to load the 32-bit twiddle factors in each layer. For the 8 butterflies that only require 1 twiddle factor, one could reduce 8 cycles at no extra cost since `ldr` has the same cost as `ldrh`.

As for the 4-layer merging strategy, due to the double size of the twiddle factors, one needs to load 15 twiddle factors into 15 FP registers with the `vldm` instruction. Compared with 8 FP registers in the original design, our design consumes 7 extra cycles. Besides, during each iteration of the first 4 layers, one needs 7 extra `vmov` instructions to retrieve the twiddle factors from the FP registers to general registers.

This extra cycle consumption can be totally covered by the cycle reduction brought by the improved Plantard arithmetic.

In order to reveal the actual effect brought by the improved modular arithmetic in our implementation, both 3-layer and 4-layer merging strategies are used in Kyber, while the better 4-layer merging strategy is adopted in NTTRU. Besides, for the implementation of NTTRU and Kyber with the 3-layer merging strategy, we use CT butterflies in NTT and GS butterflies in INTT. As for the Kyber implementation with the 4-layer merging strategy, the CT butterflies are adopted in both NTT and INTT, similar to [3].

Lazy reduction. To better illustrate the improvements the improved Plantard arithmetic brings to both CT and GS butterflies, we only discuss the lazy reduction strategies for the NTT with CT butterflies and INTT with GS butterflies here. When Montgomery arithmetic is applied to NTT and INTT, all of the coefficients will grow by q after each layer of NTT, while half of the coefficients will double after each layer of INTT. The well-known lazy reduction technique can minimize the number of modular reductions in NTT/INTT, which is mainly determined by the input range of the modular multiplication. Compared with the Montgomery arithmetic, the output range of the improved Plantard arithmetic is halved, while the input range of the improved Plantard reduction (Algorithm 21) is about 2^{α} times larger than Montgomery reduction (i.e., the reduction version Algorithm 20). Therefore, applying the improved Plantard arithmetic in NTT/INTT can halve/decrease the growing rate of coefficients compared with the one that uses Montgomery arithmetic, thus enabling better lazy reduction strategies.

CT Butterfly. Specifically, for the modulus $q = 3329$ ($9q < 2^{l-1} < 10q$) in Kyber, the forward NTT has 7 layers of butterflies. Therefore, the 7 layers of butterflies that use Montgomery multiplication will expand all of the 256 coefficients by $7q$. Therefore, at least one modular reduction is required to reduce the 256 coefficients to perform the follow-up base multiplication. On the contrary, when applying Plantard multiplication, the coefficients only grow by $3.5q$ in 7 layers. Since the inputs of

NTT are always smaller than q, $4.5q$ lies in the modular multiplication input range $[-q2^{\alpha}, q2^{\alpha}]$ of Kyber (α=3 for Kyber). Therefore, the output coefficients of NTT with Plantard multiplication can be directly used in the base multiplication without modular reduction.

It should be noted that the coefficient expansion would have a bigger effect on schemes with bigger modulus like NewHope [14] and NTTRU [82]. As for the modulus $q = 7681$ ($4q < 2^{l-1} < 5q$) in NTTRU, all of the 768 coefficients need one modular reduction after every three layers of butterflies when Montgomery multiplication is used. Since the forward NTT in NTTRU has 8 layers of butterflies, we need two modular reductions after the third and sixth layers of butterflies. The last two layers of butterflies will produce coefficients smaller than $3q$. These coefficients cannot be directly used in Montgomery multiplication because $(3q)^2 = 9q^2$, which is out of the input range of $q2^{l-1}$. Therefore, the forward NTT implemented with Montgomery arithmetic in NTTRU needs three modular reductions for 768 coefficients. However, when using the improved Plantard multiplication in NTTRU, we only need one modular reduction for 768 coefficients after the seventh-layer butterflies. The final layer of butterflies only expand the coefficients up to $1q$, which lies in the input range of $[-q2^{\alpha}, q2^{\alpha}]$ ($\alpha = 2$ for NTTRU), and these coefficients can be directly used in the base multiplication. In sum, two modular reductions for 768 coefficients are saved compared with the implementation that uses Montgomery arithmetic, which fully illustrates the benefits of a larger input range and smaller output range of our method.

GS Butterfly. As for the INTT with GS butterflies in Kyber, the advantages of applying the Plantard arithmetic are twofold. First, the maximum input value of INTT is halved after the matrix-vector multiplication (i.e., Step 6 of Algorithm 3). If one uses Montgomery arithmetic in the matrix-vector multiplication, the coefficients produced in this process would be smaller than $2q, 3q,$ and $4q$ for Kyber512, Kyber768, and Kyber1024, respectively. Therefore, one modular reduction of coefficients is required after the first and second layers of butterflies in INTT for

Kyber768/Kyber1024 and Kyber512, respectively. In our case, the modular reduction will have a one-layer delay since the matrix-vector multiplication generates coefficients smaller than $1q, 1.5q,$ and $2q$. Second, since the maximum value of the reduced coefficient is $0.5q$ after each modular reduction, 4 layers of butterflies can be conducted over the reduced coefficients instead of 3 when using Montgomery's method. After the second-layer butterflies of Kyber768/Kyber1024, the modular reduction is applied for all of the 128 coefficients; then all coefficients have a maximum value of $0.5q$. The third-layer butterflies have a step size of 8, and the first 8 coefficients ($a_0 \sim a_7$) will first grow to $8q$ after the sixth-layer butterflies. The final-layer butterflies have a step size of 128. Two sets of 8 coefficients ($a_0 \sim a_7, a_{128} \sim a_{135}$) have a maximum value of $8q$, while the rest of the coefficients are smaller than $4q$. Therefore, we only need to perform modular reduction on these 16 coefficients instead of all of the 128 coefficients in previous reports, and other coefficients will be naturally reduced by the modular multiplication with the twiddle factors and 128^{-1} in the final-layer butterflies.

The GS butterflies are also adopted for the INTT in NTTRU. The optimization brought by our method is similar to the INTT in Kyber, except that the modular reduction for half of the coefficients is required after every 3 layers of butterflies with Plantard arithmetic instead of 2 when using Montgomery arithmetic. Thus, only 2 modular reductions for 384 coefficients are required after the third and sixth layers of butterflies.

Double Plantard reduction for packed coefficients. The modular reduction of coefficients is necessary if the next operation in NTT/INTT would cause an overflow. Previous work [12] uses double Montgomery reduction or double Barrett reduction (i.e., Algorithm 10 and Algorithm 8 in [12]) to reduce two packed coefficients on Cortex-M4, which consume 7 cycles and 8 cycles, respectively. The double Barrett reduction in [12, Algorithm 8] requires two explicit shift operations for packed coefficients. Recent work further reduces its cycle counts down to 6 cycles by eliminating two explicit shift operations using the `smlawb` and `smlawt` instructions (see

Algorithm 27 Double Plantard reduction for packed coefficients

Input: A 32-bit packed integers $a = a_{top}||a_{bottom}$ where a_{top}, a_{bottom} are two 16-bit signed coefficients

Output: $r = (a_{top} \bmod^{\pm} q)||(a_{bottom} \bmod^{\pm} q)$, $-\frac{q+1}{2} \leq r_{top}, r_{bottom} < \frac{q}{2}$

1: const $\leftarrow (-2^{2l} \bmod q) \cdot (q^{-1} \bmod^{\pm} 2^{2l}) \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `smulwb` t, const, a

3: `smulwt` a, const, a

4: `smlabt` $t, t, q, q2^{\alpha}$

5: `smlabt` $a, a, q, q2^{\alpha}$

6: `pkhtb` r, a, t, asr#16

7: **return** r

[3, Algorithm 3.2]). In this work, an even better 5-cycle double Plantard reduction is proposed and shown in Algorithm 27. This algorithm is based on the fact that the Plantard reduction over a 16-bit signed integer can be viewed as a Plantard multiplication by a constant $-2^{2l} \bmod q$. And one can multiply q^{-1} by the constant $-2^{2l} \bmod q$ to obtain a precomputed constant, thus saving one multiplication by q^{-1}. Note that the input range of Algorithm 27 is the same as Algorithm 19. Since Algorithm 27 generates signed output in $[-\frac{q+1}{2}, \frac{q}{2})$, this algorithm is only applicable inside NTT/INTT. The Barrett reduction is still required outside NTT/INTT to obtain a positive result.

4.4.2 Optimized 16-bit NTT/INTT implementation on Cortex-M3 and RISC-V

Based on the faster Plantard arithmetic presented in Subsection 4.3.2, we propose a number of optimizations that are geared towards improving the efficiency and reducing the memory consumption of Kyber on Cortex-M3 and RISC-V.

Butterfly unit. To integrate the Plantard arithmetic to the butterfly unit, we follow the method described in [63] to precompute the 32-bit twiddle factor as

$\zeta = ((\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1}) \bmod^{\pm} 2^{2l}$. For two signed coefficients a and b, the CT algorithm computes $a' = a + b\zeta$ and $b' = a - b\zeta$, while the GS algorithm computes $a' = a + b$ and $b' = (a - b) \cdot \zeta$. We present the efficient implementations of CT algorithm on Cortex-M3 and RISC-V in Algorithm 28 and Algorithm 29, respectively. The instruction sequence for the GS algorithm is similar, hence we will not discuss it in detail.

We demonstrated that by utilizing the inline barrel shifter operation on Cortex-M3, one could omit the final shift operation in Algorithm 22 and achieve an efficient 5-instruction CT algorithm with the Plantard arithmetic. Conversely, the GS algorithm cannot benefit from the barrel shifter operation due to the operation difference. Specifically, GS algorithm requires one more instruction compared to the CT algorithm. Nevertheless, it is worth highlighting that both CT and GS algorithms are one instruction fewer than their Montgomery-based counterparts on Cortex-M3 [46]. Since the CT algorithm costs one instruction fewer than the GS algorithm, it motivates us to also adopt it in INTT.

Following similar optimization strategies outlined in [3] and [2], in order to minimize the side-effect of additional twists in the last layer of INTT using CT algorithm, light butterfly was proposed and applied in the first three layers of INTT. The so-called light butterfly is to reduce the modular multiplication by the twiddle factor in the CT butterfly, i.e., only computing $a' = a + b$ and $b' = a - b$. Additionally, they suggested that using CT algorithm in INTT could also help to decrease the number of modular reductions and yield a superior INTT implementation on Cortex-M4 [3]. After applying the CT algorithm to INTT of Kyber, we further validate its extensibility on Cortex-M3. It is worth to note that we are the first to apply the above improved CT algorithm to INTT on Cortex-M3.

Algorithm 29 demonstrates the instruction sequence of the CT algorithm on RISC-V. Unlike the implementation on Cortex-M3, the GS and CT algorithms are equally efficient on RISC-V. Initially, we also contemplated the use of CT algorithm in INTT on RISC-V; however, the results indicated that the INTT with CT algo-

Algorithm 28 CT algorithm on Cortex-M3

Input: Two signed integers a, b, the 32-bit twiddle factor ζ

Output: $a' = a + b\zeta, b' = a - b\zeta$

1: `mul` b, b, ζ

2: `add` $b, 2^{\alpha}, b,$ asr #16

3: `mul` t, b, q

4: `sub` $b', a, t,$ asr #16

5: `add` $a', a, t,$ asr #16

6: **return** a', b'

rithm cannot outperform the GS one. The crucial reasons for that are two-fold. Firstly, the use of CT algorithm in INTT results in the doubling of the twiddle factors, which increases the memory access overhead. Specifically, in our implementation, the INTT with CT algorithm consumes 61 additional memory accesses for the twiddle factors, which reduces its efficiency and warrants a further 0.5KiB memory to accommodate these extra twiddle factors. The extra memory cost also increases the difficulty of deploying Kyber on memory-constrained platforms. Secondly, the number of modular reduction in INTT with GS algorithm has been significantly reduced by using Algorithm 17 on RISC-V (see the better lazy reduction strategy in Section 4.4.2 for detailed discussions). Thus, there left only a small space for further optimization through the utilization of CT algorithm in INTT.

In summary, the optimal combination of butterfly usage on Cortex-M3 is to apply CT algorithm on both NTT and INTT. On the other hand, the most effective pattern on RISC-V is to use CT algorithm on NTT but GS algorithm on INTT.

Layer merging. Layer merging is an essential optimization strategy in NTT/INTT when memory access operations are costly. This strategy effectively reduces the expensive memory access by loading multiple coefficients at once and processing more NTT/INTT layers over these coefficients. The design of optimal layer merging strategy depends on the number of available registers on the target platform.

Algorithm 29 CT algorithm on RISC-V

Input: Two signed integers a, b, the 32-bit twiddle factor ζ

Output: $a' = a + b\zeta, b' = a - b\zeta$

1: `mul` b, b, ζ

2: `srai` $b, b,$ #16

3: `addi` $b, b, 2^{\alpha}$

4: `mulh` $t, b, q2^{l}$

5: `sub` b', a, t

6: `add` a', a, t

7: **return** a', b'

For example, fourteen registers are programming-available on Cortex-M3. Thus, a maximum of eight coefficients can be loaded at once, allowing to merge three layers of NTT/INTT on this platform. The remaining six registers are utilized to store addresses or execute the modular arithmetic in the butterfly unit. On RISC-V, an abundance of 30 programming-available registers are provided with greater flexibility. Therefore, sixteen coefficients can be loaded at once, allowing to merge four layers of NTT/INTT at once, and there are still 14 registers left for other usage. Consequently, the 4-layer merging strategy is better on RISC-V.

In summary, for the 7-layer NTT/INTT in Kyber, we adopt the 3+3+1 and 3+1+3 layer merging strategy for NTT and INTT, respectively, on Cortex-M3. The motivation behind the choice of the 3+1+3 layer merging strategy instead of 3+3+1 will be discussed in Section 4.4.2. On the other hand, the 4+3 and 3+4 layer merging strategy are used for NTT and INTT on RISC-V, respectively.

Better lazy reduction. The so-called "lazy reduction strategy" means deferring the modular reduction of coefficients until the value of coefficients exceed the maximum bound of the register or the input range of the subsequent operation. The input range of the modular multiplication by a constant in the butterfly unit is particularly crucial in facilitating this strategy. The NTT implementation with CT algorithm al-

gorithm using Plantard arithmetic only increases the coefficients by $3.5q$. The large input range and small output range of Plantard arithmetic enables us to eliminate all the modular reductions of coefficients in NTT on Cortex-M3 and RISC-V. This is a noteworthy improvement compared to the Montgomery-based NTT implementation, which requires one modular reduction for all coefficients. Therefore, the focus of this study is on the exploration of better lazy reduction strategy for INTT on Cortex-M3 and RISC-V.

An even better lazy reduction strategy is designed for INTT in this work, compared to the version on Cortex-M4 in [63]. Two main factors contribute to this new design. Firstly, Huang et al. [63] utilized the SIMD extension of Cortex-M4 to speed up NTT/INTT; thus the coefficient in INTT cannot exceed the bound of a 16-bit signed integer. For Kyber's modulus $q = 3329$, the coefficients must stay within $[-9.5q, 9.5q]$. However, on the low-end 32-bit platforms that lack of SIMD extension, each coefficient is loaded to a 32-bit register, which has the potential to temporarily overflow 16-bit signed integer range before it is stored back to a 16-bit memory. Secondly, the input range of the Plantard arithmetic is enlarged from $[-64q, 64q]$ in [63, Algorithm 11] to $[-137q, 230q]$, which allows for a "lazier" modular reduction.

Input polynomial of INTT. Before moving to the detailed strategy, let's analyze the coefficient size of the input polynomial of INTT first. It is worth noting that matrix-vector multiplication and vector inner product are the two operations that could expand the coefficient size of INTT's input polynomial. The resulting polynomial of these two operations must add up k intermediate polynomials, where k equals $2, 3, 4$ for the three Kyber variants, respectively. Abdulrahman et al. [3] proposed two versions of implementations for these two operations and its underlying pointwise multiplication: the stack-version and the speed-version. In this study, we apply the Plantard arithmetic to both versions of implementations.

- The stack-version process involves reducing each intermediate polynomial and accumulating k reduced polynomials together, which generates an "unreduced"

polynomial (i.e., every coefficient of the polynomial is beyond the output range of modular reduction). When implemented with the Plantard arithmetic, the stack-version matrix-vector product and vector inner product produce coefficients within the range $(-kq/2, kq/2)$.

- The speed-version implementation is built on top of the stack-version implementation. An intermediate 32-bit array is used to cache the sum of k intermediate polynomials, and this intermediate result is reduced after the accumulation process, producing a "reduced" polynomial (i.e., every coefficient is in the output range of modular reduction). When implemented with the Plantard arithmetic, the speed-version code generates coefficients in the range $(-q/2, q/2)$.

Better lazy reduction on RISC-V. We first introduce a better lazy reduction strategy for INTT on RISC-V. Since INTT is computed by using GS algorithm in combination with the 3+4 layer merging strategy on RISC-V, after the third layer of INTT, the coefficients have to be stored back to 16-bit signed integers in memory. Therefore, we need to analyze the coefficient size to ensure that they can fit in 16-bit signed integer after the third layer of INTT.

Figure 4.1 demonstrates the first 16 coefficients of the first three layers of a length-128 INTT implemented via GS algorithm. Since the input coefficients of the stack-version INTT are within the bound $(-kq/2, kq/2)$, we let the red number x in Figure 4.1 equal to $kq/2$, which is the largest absolute value of the input range. After three layers of GS butterflies computation, the first 2 coefficients (out of 16 coefficients) are expanded up to $8x$, namely $4kq$. In the case of Kyber512 ($k = 2$), the term $4kq$ is equal to $8q$, and it does not overflow 16-bit signed integer after the third layer. The later four layers of INTT produce coefficients that reach up to $128q$, which exceeds the input range of the Plantard arithmetic in [63, Algorithm 11] $[-64q, 64q]$, but remains within the input range of the proposed Plantard arithmetic with enlarged input range $[-137q, 230q]$. These coefficients will be reduced by the

modular multiplication with the twiddle factors and 128^{-1} in the last layer of INTT. Hence, modular reduction can be completely avoided in the INTT of Kyber512. As for Kyber768 and Kyber1024, where $k = 3$ and $k = 4$ respectively, the term $4kq$ is equal to $12q$ and $16q$, respectively. Both $12q$ and $16q$ overflow a 16-bit signed integer. Therefore, modular reductions are needed for these coefficients (2 out of 16). Due to the symmetric property of INTT, the entire INTT for Kyber768 and Kyber1024 requires modular reductions for $256 \times 2/16 = 32$ coefficients only. By using the 3+4 layer merging strategy, the last four layers of INTT only expand some coefficients, e.g., a_2 and a_3 in Figure 4.1, from $2q$ up to $32q$. These coefficients will also be reduced by the modular multiplication with the twiddle factors and 128^{-1} in the last layer of INTT. Therefore, no further modular reduction is needed.

The input coefficients of the speed-version INTT are within the bound $(-q/2, q/2)$ for all three Kyber variants. We let the red number x in Figure 4.1 equal to $q/2$, which is the largest absolute value of the input range. After three layers' computation, the absolute value of these coefficients are smaller than $8x = 4q$, which is well below the maximum value of a 16-bit signed integer and can be securely stored back to the 16-bit memory. After seven layers of INTT computation by using GS algorithm, these coefficients are expanded to $64q$. However, this is still within the input range of the proposed Plantard arithmetic. Thus, we are able to eliminate all the modular reduction of the speed-version INTT for the three Kyber variants.

Better lazy reduction on Cortex-M3. We now describe a better lazy reduction strategy for INTT on Cortex-M3. The INTT on Cortex-M3 is computed by using CT algorithm in combination with the 3+1+3 layer merging strategy. After the third and fourth layer of INTT, the coefficients have to be stored back to 16-bit signed integers in memory. Therefore, we need to analyze the coefficient size and ensure that they can fit in 16-bit signed integer after the third and fourth layer.

Figure 4.2 illustrates the first 8 coefficients of the first three layers of a length-128 INTT by using light butterfly and CT algorithm. As shown in the topmost dashed rectangles of Figure 4.2, the light butterfly proposed in [3] only involves one addition

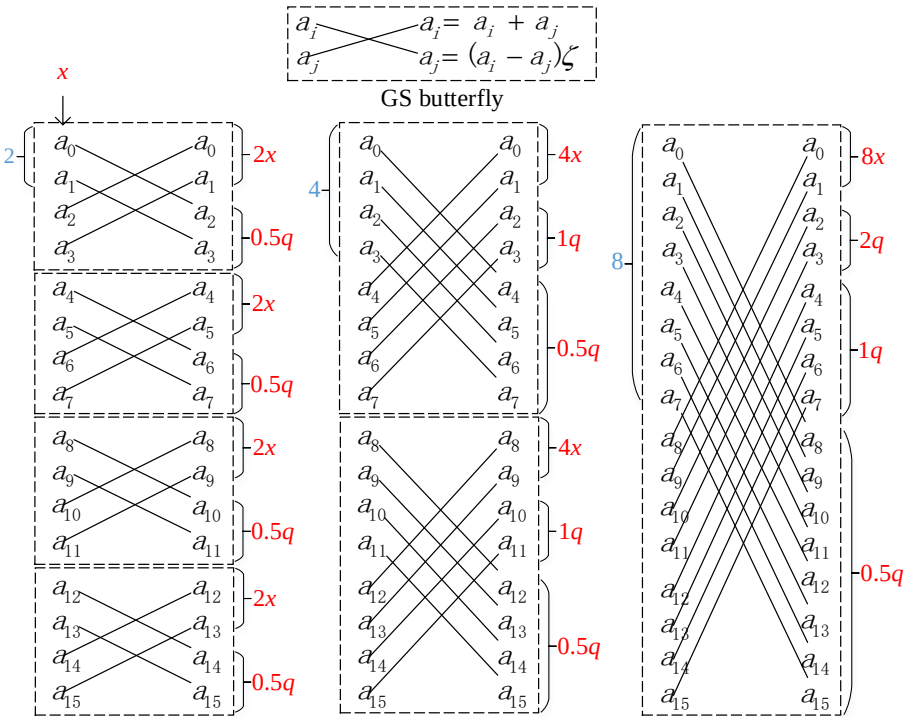


Figure 4.1: Example of the first 16 coefficients of the first three layers of a length-128 INTT using GS algorithm on RISC-V.

Note: a_i and a_j represent coefficients of polynomial a. The red number x in upper left corner represents the maximum absolute value of the input coefficient of INTT. Dashed rectangle represents GS butterflies of various step size; the computation details of GS butterfly are described in the topmost dashed rectangle. The blue number on the left-hand side of the rectangle indicates the step size of the butterfly unit, while the red number represents the maximum absolute value of the corresponding coefficients after the computation of each layer.

and one subtraction, omitting one modular multiplication by the twiddle factor of the CT algorithm. Consequently, each light butterfly will double the size of both coefficients while each CT butterfly will only increase each coefficient by $0.5q$. For the stack-version INTT implementation, the absolute value of the input coefficients is smaller than $kq/2$. If we let the red number x in Figure 4.2 equal to $kq/2$, the computation of three layers INTT would expand $256 \times 2/8 = 64$ coefficients (e.g., a_0 and a_4 in Figure 4.2) up to $8x = 4kq$. Similar to the analysis in Section Section 4.4.2, the term $4kq$ overflows a 16-bit signed integer for Kyber768 and Kyber1024. There-

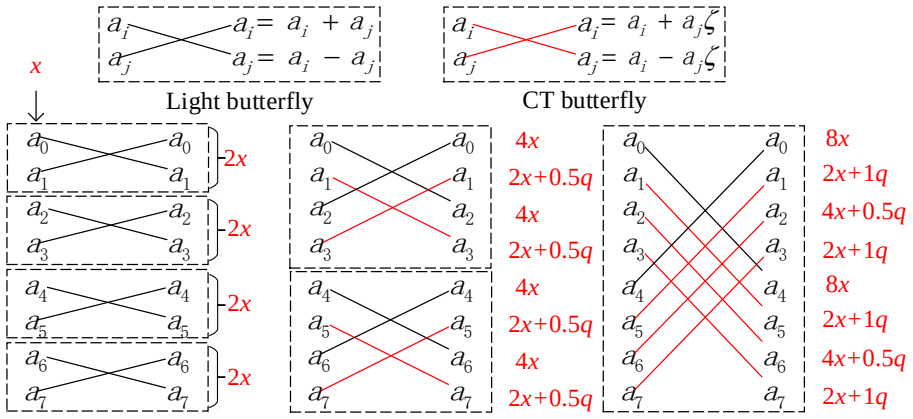


Figure 4.2: Example of the first 8 coefficients of the first three layers of a length-128 INTT by using CT algorithm on Cortex-M3.

Note: a_i and a_j represent coefficients of the polynomial a. The red number x in upper left corner represents the maximum absolute value of the input coefficient of INTT. Dashed rectangle represents light or CT butterflies of various step size; the computation details of light butterfly and CT algorithm are described in the topmost two dashed rectangles; the butterflies with black crossing line denote light butterfly while the butterflies with red crossing line represent CT algorithm; the red number represents the maximum absolute value of the corresponding coefficients after the computation of each layer.

fore, one might need modular reductions for 64 coefficients after the third layer of INTT. However, in order to minimize the number of modular reductions, we curtail half of the modular reductions by performing modular reductions for $256 \times 1/8 = 32$ coefficients (e.g., a_0 in Figure 4.2) after the second layer of INTT. After that, a_0 is reduced down to $0.5q$, and the third layer of INTT would only expand a_0 and a_4 in Figure 4.2 up to $4x + 0.5q$, which is equal to $6.5q/8.5q$ for Kyber768/Kyber1024. These numbers could fit in 16-bit signed integers. Since we use the 3+1+3 layer merging strategy and adopt the CT algorithm in the fourth layer, the fourth layer of INTT with CT algorithm would only increase the coefficients by $0.5q$. Therefore, the maximum absolute value of coefficients is $9q$ after the fourth layer, which can still fit in a 16-bit signed integer. The reason why we adopt the 3+1+3 layer merging strategy instead of 3+3+1 is that it limits the growth of coefficients and obviates the

need for more modular reductions. If we adopt the 3+3+1 layer merging strategy, the second three layers of INTT with CT algorithm would expand $256 \times 4/8 = 128$ coefficients (e.g., a_0, a_2, a_4 and a_6 in Figure 4.2) up to $10q$, which will overflow a 16-bit signed integer, thus necessitating more modular reductions.

The final three layers of INTT in the 3+1+3 layer merging strategy increase the coefficients up from $9q$ to $72q$. It should be noted that $72q$ likewise surpasses the input range of $[-64q, 64q]$ in Plantard arithmetic [63, Algorithm 11], but can still fit into the input range $[-137q, 230q]$ of the proposed Plantard arithmetic. This further demonstrates the contribution of the larger input range to the better lazy reduction strategy. Similar to the INTT using GS algorithm on RISC-V, we also eliminate the modular reduction of coefficients for the stack-version INTT of Kyber512 and speed-version INTT of all three Kyber variants on Cortex-M3.

Overall, we only need modular reduction for 32 coefficients in the stack-version INTT of Kyber768 and Kyber1024 on both platforms. Conversely, we entirely eliminate the modular reduction of coefficients in the stack-version INTT of Kyber512 and the speed-version INTT of all three Kyber variants. The better lazy reduction strategy is made possible by leveraging the excellent merits of the Plantard arithmetic, the optimal layer merging strategies, and the 32-bit platforms' properties.

4.4.3 Pointwise multiplication and memory optimizations

Pointwise multiplication. After the 7-layer NTT transform, two polynomials a and b are transformed into their NTT domain $\hat{a}$ and $\hat{b}$. The pointwise multiplication $\hat{c} = \hat{a} \circ \hat{b}$ is performed over $\mathbb{Z}_q[X]/(X^2 - \zeta^{2\mathrm{br}_7(i)+1})$. The symbol $\mathrm{br}_7(i)$ denotes the bit reversal operation of a 7-bit integer $i \in [0, 127]$ to obtain the corresponding twiddle factor. The symbol $\circ$ consists of 128 pointwise multiplications, and each of them is performed as $\hat{c}_{2i} + \hat{c}_{2i+1}X = (\hat{a}_{2i} + \hat{a}_{2i+1}X)(\hat{b}_{2i} + \hat{b}_{2i+1}X) \bmod (X^2 - \zeta^{2\mathrm{br}_7(i)+1})$, where $\hat{c}_{2i} = \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ and $\hat{c}_{2i+1} = \hat{a}_{2i}\hat{b}_{2i+1} + \hat{b}_{2i}\hat{a}_{2i+1}$. As mentioned in Section 4.4.2, to align with the stack-version and speed-version matrix-vector multiplication, we also implement two versions of pointwise multiplications

for Kyber.

For the stack-version pointwise multiplication, we follow the lazy reduction strategy proposed in [12], namely $\hat{c}_{2i} = (\hat{a}_{2i} \cdot \hat{b}_{2i} + \hat{a}_{2i+1} \cdot (\hat{b}_{2i+1} \cdot \zeta^{2\text{br}_7(i)+1} \bmod q)) \bmod q$ and $\hat{c}_{2i+1} = (\hat{a}_{2i} \cdot \hat{b}_{2i+1} + \hat{b}_{2i} \cdot \hat{a}_{2i+1}) \bmod q$. In total, three modular reductions are required for each pointwise multiplication and the result is stored in 16-bit array. We can see that the computation of $\hat{b}_{2i+1} \cdot \zeta^{2\text{br}_7(i)+1} \bmod q$ is a modular multiplication by a twiddle factor $\zeta^{2\text{br}_7(i)+1}$ and can be speeded up with the efficient Plantard multiplication by a constant (Algorithm 22 and Algorithm 23). Overall, compared to the Montgomery-based implementation, we can utilize the Plantard arithmetic to reduce 128 multiplications in the stack-version pointwise multiplication implementation. The other two modular reductions, on the other hand, can only be implemented with the Plantard reduction (see Section 4.3.2). Note that the proposed Plantard reduction is as efficient as the state-of-the-art Montgomery and Barrett reduction on the target platforms. After the Plantard reduction, the coefficient range of each pointwise multiplication lies in $[-\frac{q+1}{2}, \frac{q}{2})$.

As for the speed-version implementation, we follow the asymmetric multiplication and a better accumulation strategies in [3] and [2], which help to further minimize the number of modular multiplications or modular reductions, with nearly doubled the stack usage. The so-called asymmetric multiplication is that during the matrix-vector product $\mathbf{As}$ in Kyber, they [3, 2] observed that every row of the NTT-domain matrix $\hat{\mathbf{A}}$ needs to multiply the NTT-domain secret vector $\hat{\mathbf{s}}$ (vector inner product) for k times. For the pointwise multiplication of two NTT-domain polynomials $\hat{a} \circ \hat{s} = \hat{c}$, the term $\hat{s}_{2i+1}\zeta^{2\text{br}_7(i)+1}$ also needs to be computed k times throughout the process. Therefore, they proposed to reduce these computations by caching $\hat{s}_{2i+1}\zeta^{2\text{br}_7(i)+1}$ using an additional polynomial vector $\hat{\mathbf{s}}'$. When $\hat{\mathbf{s}}$ is multiplied with the first row of matrix $\hat{\mathbf{A}}$, the term $\hat{s}_{2i+1} \cdot \zeta^{2\text{br}_7(i)+1} \bmod q$ for each polynomial of the vector $\hat{\mathbf{s}}$ is computed and stored in $\hat{\mathbf{s}}'$. Similar to the stack-version implementation, this computation can also be speeded up with the efficient Plantard multiplication by a constant. The follow-up vector inner products of $\hat{\mathbf{s}}$ with the

remaining $k-1$ rows of matrix $\hat{\mathbf{A}}$ can directly load the corresponding $\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ from $\hat{\mathbf{s}}'$ without re-computing the modular multiplication.

Similarly, the better accumulation technique also trades the speed with extra memory. This technique is utilized in the vector inner product, i.e. $c = \hat{\mathbf{a}} \times \hat{\mathbf{s}} = \sum_{i=0}^{k-1} \hat{a}_i \circ \hat{s}_i, k \in \{2,3,4\}$, which requires to accumulate k pointwise multiplications. Abdulrahman et al. [3, Section 3.4] showed that one can reduce the final modular reduction of c_{2i} and c_{2i+1}, namely only computing $\hat{c}_{2i} = \hat{a}_{2i} \cdot \hat{b}_{2i} + \hat{a}_{2i+1} \cdot (\hat{b}_{2i+1} \cdot \zeta^{2\mathrm{br}_7(i)+1} \bmod q)$ and $\hat{c}_{2i+1} = \hat{a}_{2i} \cdot \hat{b}_{2i+1} + \hat{b}_{2i} \cdot \hat{a}_{2i+1}$, and then accumulate these k pointwise multiplications together in a 32-bit array with 256 entries without overflowing 32-bit signed integers. The k-th pointwise multiplication will finally reduce each 32-bit coefficient down to 16-bit using the Plantard reduction. After that, the coefficient range of the vector inner product lies in $[-\frac{q+1}{2}, \frac{q}{2})$.

Memory optimizations on Cortex-M3 and RISC-V. The speed-version Kyber implementation from [3] utilizes the asymmetric multiplication and better accumulation techniques in [2] to minimize the number of modular multiplications or modular reductions. However, these techniques double the stack usage and pose a challenge for its deployment on memory-constrained IoT devices. In this paper, we propose two memory optimized techniques for the speed-version implementation to address this concern and make it more feasible for such devices.

The first memory optimized technique is carried out for the asymmetric multiplication technique. Recall that Abdulrahman et al. [3, 2] proposed to cache the term $\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ using an additional polynomial vector $\hat{\mathbf{s}}'$. To make use of the SIMD instruction `smuad` on Cortex-M4, they also stored a redundant $\hat{s}_{2i}$ together with each $\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ in $\hat{\mathbf{s}}'$. This leads to the increment of memory usage, as the redundant $\hat{s}_{2i}$ term has already existed in the original polynomial vector $\hat{\mathbf{s}}$. In this work, we show that the extra memory space is unnecessary on low-end 32-bit platforms. We propose using *poly_half* and *polyvec_half* to cache $\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ only, where each *poly_half* consists of 128 instead of 256 coefficients and each *polyvec_half* consists of k *poly_half*. Our approach results in a halving of stack usage to cache the temporary

terms. Specifically, by using the *polyvec_half* to cache all terms $\hat{s}_{2i+1}\zeta^{2\text{br}_7(i)+1}$ for one secret vector in the speed-version code, we could reduce the stack usage by 0.5KiB, 0.75KiB, and 1KiB, in three PKC protocols of Kyber512, Kyber768, and Kyber1024, respectively.

In addition to the memory optimized technique described above, we propose another approach to reduce the memory usage for the better accumulation technique used in the IND-CPA key generation protocol of Kyber, which utilizes a temporary 32-bit array with 256 entries to store the intermediate accumulation result of the vector inner product. We observe that in the IND-CPA key generation protocol, the memory space reserved for the 32-bit temporary array and polynomial *pkp* can be merged, where *pkp* is a temporary polynomial in the key generation protocol. We introduce a new data structure called *poly_double* that can hold 512 16-bit coefficients, which has the same memory footprint as the temporary 32-bit array. By leveraging data type conversion, we can re-purpose the memory allocated for the temporary array to store *pkp*, reducing its memory usage by 0.5KiB in the key generation protocol.

Together, these two memory optimizations improve the feasibility of the speed-version implementation on memory-constrained IoT devices significantly, such as the selected 16KiB SiFive board. Notably, while the second approach only applies to the KEM key generation protocol, the first technique can be applied to all three KEM protocols of Kyber.

4.5 Dilithium optimizations on ARMv7-M

This section presents the small NTT (i.e., 16-bit NTT) for $c\mathbf{s}_i$ in Dilithium on Cortex-M3 and M4, as well as the multi-moduli NTT for $c\mathbf{t}_0$ in Dilithium on Cortex-M3. Then, the efficient 16-bit NTT and explicit CRT implementations with the Plantard arithmetic are introduced. Finally, the security and extensibility of this work will be briefly discussed.

4.5.1 Small NTTs for $c\mathbf{s}_i$ on Cortex-M3 and M4

The polynomial multiplication of $c\mathbf{s}_i$ produces a polynomial with a coefficient range smaller than $\beta = \tau\eta$. Hence, the polynomial multiplication of $c\mathbf{s}_i$ can be treated as polynomial multiplication over the NTT-friendly polynomial ring $\mathbb{Z}_{q'}[X]/(X^n + 1)$ with $q' > 2\beta$ [36, Section 2.4.6]. The coefficient range β equals 78, 196 and 120 for all three Dilithium parameter sets (see Table 5.1), respectively. Based on this observation, previous work [3] implemented $c\mathbf{s}_i$ with the small NTT over 769 ($769 > 2 \times 196$) in Dilithium3 and FNT over 257 ($257 > 2 \times 120$) in Dilithium2 and Dilithium5 on Cortex-M4. Since the pointwise multiplication and INTT are the core computations during the rejection sampling in `sign`, the FNT over 257 implementation in [3] provides faster pointwise multiplication, which enables better Dilithium2 and Dilithium5 than the NTT over 769. Therefore, we also reuse the FNT over 257 for Dilithium2 and Dilithium5 on Cortex-M4. The NTT over 769 with Plantard arithmetic is adopted for Dilithium3 on Cortex-M4.

On Cortex-M3, we decide to reuse the NTT over 769 to compute $c\mathbf{s}_i$ for all three Dilithium parameter sets for the following reasons. (1) The 16-bit NTT over 769 can be speeded up with the efficient Plantard arithmetic while the FNT over 257 is already very efficient, and there is not much space for further optimization. (2) We will later show that the NTT over 769 together with NTT over 3329 can enable an efficient multi-moduli NTT implementation on Cortex-M3 (see Subsection 4.5.2). Therefore, the NTT over 769 can be reused for computing both $c\mathbf{s}_i$ and $c\mathbf{t}_i$. Although the FNT over 257 can also enable the multi-moduli NTT together with the NTT over 7681, the FNT over 257 is only applicable for Dilithium2 and Dilithium5 and we would need two different multi-moduli NTT implementations for all variants of Dilithium. On the other hand, the NTT over 769 can be reused for all three Dilithium variants, and using the NTT over 769 allows us to reuse one multi-moduli NTT implementation for all three variants of Dilithium.

4.5.2 Multi-moduli NTT for $c\mathbf{t}_i$ on Cortex-M3

Parameters choice. As for $c\mathbf{t}_i$, the coefficient range of the product is smaller than $\tau 2^{10}$ or $\tau 2^{12}$, and the maximum value of τ is 60 according to Table 5.1. Therefore, the coefficient range of $c\mathbf{t}_i$ is $\beta' = \tau 2^{12} = 60 \times 2^{12} = 245760$. These polynomial multiplications can be treated as the polynomial multiplication over $\mathbb{Z}_{q'}/[X](X^n+1)$ with a 32-bit NTT-friendly modulus $q' > 2\beta' = 491520$ [36, Section 2.4.6]. As mentioned in Subsection 2.4.4, either the variable-time or constant-time 32-bit NTT is much slower than 16-bit NTT on Cortex-M3. Recent research [65] shows that the 16-bit NTT can be further optimized using the efficient Plantard arithmetic on Cortex-M3. The 16-bit NTT with Plantard arithmetic in [65] is 2.41× faster than the variable-time 32-bit NTT and 4.11× faster than the constant-time 32-bit NTT on Cortex-M3.

Based on this observation, instead of choosing a 32-bit NTT-friendly prime and directly using the slow 32-bit NTT for computing $c\mathbf{t}_i$, we choose a composite modulus $q' = 769 \times 3329 = 2560001$ defined by the product of two 16-bit NTT-friendly primes 769 and 3329. It is easy to see that the selected modulus q' is larger than $2\beta'$; so the polynomial multiplication of $c\mathbf{t}_i$ can be treated as a multiplication over $\mathbb{Z}_{q'}[X]/(X^n+1)$ with $q' = q_0 \times q_1, q_0 = 769$ and $q_1 = 3329$. Using this composite modulus, we could utilize the multi-moduli NTT technique to speed up $c\mathbf{t}_i$ in Dilithium. It should be noted that this technique has been adopted for Saber, NTRU and LAC [36, 2] but has not been applied to Dilithium yet. Previous work [57] also briefly discussed the potential to replace the slow 32-bit NTT with multi-moduli NTT, but they claimed that the multi-moduli NTT implementation with Montgomery arithmetic was slower than their schoolbook-based 32-bit NTT [57, Section 4.1]. This paper revisits this idea and shows that together with the efficient Plantard arithmetic for 16-bit modulus, we are able to speed up the multi-moduli NTT and utilize it to compute $c\mathbf{t}_0$ on Cortex-M3.

Incomplete NTTs over 769 and 3329. Unlike Dilithium where the parameters q and n satisfy $2n|(q-1)$, the moduli $q_0 = 769$ and $q_1 = 3329$ only satisfy $n|(q_i - 1)$ for $i = 0, 1$. That means there exists an n-th root of unity ζ_i for the corresponding q_i such that the cyclotomic polynomial $X^n + 1$ can be factorized to degree-1 factors $X^2 - \zeta_i^j$ with $j = 1, 3, 5, \ldots, 255$. According to CRT, the polynomial ring R_{q_i} is isomorphic to the product of the degree-1 polynomial rings, namely $R_{q_i} \cong \prod \mathbb{Z}_{q_i}[X]/(X^2 - \zeta_i^j)$. Using the incomplete NTTs, the performance of NTT and INTT can be improved because they only need to compute 7 layers of NTT or INTT. On the other hand, the pointwise multiplications for these incomplete NTTs are performed modulo $X^2 - \zeta_i^j$ which is a little more complicated than pointwise multiplication modulo the degree-0 $X - \zeta$ in Dilithium. Nevertheless, these pointwise multiplications have been greatly improved by using lazy reduction [12] and asymmetric multiplication strategies [2, 3]. Overall, using the incomplete NTTs could still obtain performance improvement, which can be demonstrated by the parameters choice of Kyber [31].

NTT with composite modulus. In this paper, we are dealing with NTT over the composite modulus $q' = q_0 q_1$ with $q_0 = 769$ and $q_1 = 3329$. Let ζ_0 and ζ_1 be the 256-th roots of unity in $\mathbb{Z}_{q_0}$ and $\mathbb{Z}_{q_1}$, respectively. According to the CRT and incomplete NTTs, we have the following isomorphism (the proof is similar to [2, Appendix E]):

$$\begin{aligned}
&\mathbb{Z}_{q_0 q_1} \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1}; \\
&\mathbb{Z}_{q_0}[X]/(X^{256} + 1) \cong \mathbb{Z}_{q_0}[X]/(X^2 - \zeta_0^j), j = 1, 3, 5, \ldots, 255; \\
&\mathbb{Z}_{q_1}[X]/(X^{256} + 1) \cong \mathbb{Z}_{q_1}[X]/(X^2 - \zeta_1^j), j = 1, 3, 5, \ldots, 255;
\end{aligned} \tag{4.5.1}$$

Overall, the polynomial multiplication over $\mathbb{Z}_{q'}[X]/(X^n + 1)$ can be separately computed using the faster 16-bit NTTs over $R_{q_i} = \mathbb{Z}_{q_i}[X]/(X^n + 1)$ [2, Section 4.2.1]. After the 16-bit NTTs, pointwise multiplications and INTTs, one then uses the explicit CRT described in Subsection 2.4.5 to reconstruct the 32-bit results.

Algorithm 30 Multi-moduli NTT for computing 32-bit NTT on Cortex-M3

Input: Declare arrays: `int32_t c_32[256],t_32[256],tmp_32[256],res_32[256]`

Input: Declare pointers: `int16_t *cl_16=(int16_t*)c_32; int16_t *ch_16=(int16_t*)&c_32[128]; int16_t *tl_16=(int16_t*)t_32; int16_t *th_16=(int16_t*)&t_32[128]; int16_t *tmpl_16=(int16_t*)tmp_32; int16_t *tmph_16=(int16_t*)&tmp_32[128];`

1: `cl_16[256]` $\leftarrow c$, `ch_16[256]` $\leftarrow c$ ▷ Pre-store c in the bottom and top halves of `c_32` as 16-bit arrays

2: `tl_16[256]` $\leftarrow t$, `th_16[256]` $\leftarrow t$ ▷ Pre-store t in the bottom and top halves of `t_32` as 16-bit arrays

3: `cl_16[256]` $= \mathrm{NTT}_{q_0}($`cl_16`$)$ ▷ $\hat{c}_0 = \mathrm{NTT}_{q_0}(c)$

4: `ch_16[256]` $= \mathrm{NTT}_{q_1}($`ch_16`$)$ ▷ $\hat{c}_1 = \mathrm{NTT}_{q_1}(c)$

5: `tl_16[256]` $= \mathrm{NTT}_{q_0}($`tl_16`$)$ ▷ $\hat{t}_0 = \mathrm{NTT}_{q_0}(t)$

6: `th_16[256]` $= \mathrm{NTT}_{q_1}($`th_16`$)$ ▷ $\hat{t}_1 = \mathrm{NTT}_{q_1}(t)$

7: `tmpl_16[256]` $= \mathrm{basemul}_{q_0}($`cl_16`$,$`tl_16`$)$ ▷ $\hat{c}_0 \cdot \hat{t}_0 = \mathrm{basemul}_{q_0}(\hat{c}_0, \hat{t}_0)$

8: `tmph_16[256]` $= \mathrm{basemul}_{q_1}($`ch_16`$,$`th_16`$)$ ▷ $\hat{c}_1 \cdot \hat{t}_1 = \mathrm{basemul}_{q_1}(\hat{c}_1, \hat{t}_1)$

9: `tmpl_16[256]` $= \mathrm{INTT}_{q_0}($`tmpl_16`$)$ ▷ $\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0)$

10: `tmph_16[256]` $= \mathrm{INTT}_{q_1}($`tmph_16`$)$ ▷ $\mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1)$

11: `res_32[256]` $= \mathrm{CRT}($`tmpl_16`$,$`tmph_16`$)$ ▷ $\mathrm{CRT}(\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0), \mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1))$

12: **return** `res_32`

Memory layout. The workload of the multi-moduli NTT is illustrated in Algorithm 30. To clearly understand the workload, we need to describe the memory layout for the multi-moduli NTT. Let t be a polynomial in $\mathbf{t}_i$ with $i = 0, 1$. To compute the polynomial multiplication ct, previous implementation using 32-bit NTT [57] stores c and t in 32-bit arrays, i.e., `int32_t c_32[256]`, `t_32[256]`. However, according to the coefficient range of c and t (see Table 5.1), they can actually be stored in 16-bit arrays. In order to utilize the multi-moduli NTT to compute ct, we modify the `unpack_sk` or `poly_challenge` functions to generate two copies of c and t and pre-store them as 16-bit arrays in the bottom and top halves of `c_32`

and t_32. We use 16-bit pointers (int16_t *cl_16,*ch_16,*tl_16,*th_16) to access the bottom and top halves of these 32-bit arrays as 16-bit arrays. Overall, as shown in Algorithm 30, the multi-moduli NTT for ct needs to compute four NTTs, two pointwise multiplications, two INTTs, and one CRT. Compared to the 32-bit NTT implementation, our multi-moduli NTT implementation requires 1024 bytes of memory (tmp_32) for the pointwise multiplication and CRT computation. However, when compared to a very similar implementation for Saber in [2, Algorithm 5], our implementation only needs one-third of their memory. The main difference is that we can reuse the memory of c and t for in-place NTT/INTT implementation using the new unpack_sk and poly_challenge functions; thus achieve significant memory savings.

Multi-moduli NTT in sign and verify. The proposed multi-moduli NTT is applicable to $c\mathbf{t}_0$ in sign (Algorithm 9) and $c\mathbf{t}_1$ in verify (Algorithm 8) on Cortex-M3. The present multi-moduli NTT incorporates two NTTs over 769 and 3329. Notably, the NTT over 769 is also utilized for the computation of $c\mathbf{s}_i$ in sign. Previous work [3] utilized the FNT over 257 or NTT over 769 for $c\mathbf{s}_i$ on Cortex-M4. However, their $c\mathbf{t}_0$ in sign was implemented with the 32-bit NTT of Dilithium. Consequently, they needed to perform both a 32-bit NTT and a 16-bit NTT on c for the subsequent computations. In contrast, by employing multi-moduli NTT over the chosen modulus $q' = q_0q_1$, we only require one multi-moduli NTT operation on c. The computation of $c\mathbf{s}_i$ can leverage the NTT-domain $\hat{c}$ over q_0 generated in the multi-moduli NTT, thereby eliminating an extra NTT computation on c.

Although the multi-moduli NTT is beneficial in sign, it is somewhat impractical in verify. One of the main step in verify is Step 4 in Algorithm 8, i.e. $\mathbf{Az} - c\mathbf{t}_1 \times 2^d$. It is recommended to perform the INTT after the subtraction. When using the 32-bit NTT for $c\mathbf{t}_1 \times 2^d$, the subtraction can be carried out after the pointwise multiplication, as both operations utilize the same NTT over the modulus q. However, when employing a multi-moduli NTT over $q' = q_0q_1$ for $c\mathbf{t}_1$ and a 32-bit NTT over q for $\mathbf{Az}$, they are in different NTT domains and can not be directly

subtracted. Consequently, using the multi-moduli NTT in `verify` necessitates an additional INTT computation for a polynomial vector. Furthermore, since c and $\mathbf{t}_1$ are public, previous implementation on Cortex-M3 [57] chooses to utilize variable-time 32-bit NTT, which is faster than constant-time one. Although our multi-moduli NTT is faster than the variable-time 32-bit NTT (see Section 4.7), the extra INTT computation for a polynomial vector outweighs the benefits of using the multi-moduli NTT. Therefore, the proposed multi-moduli NTT implementation offers advantages solely in accelerating the signature generation process of Dilithium on Cortex-M3.

4.5.3 Faster 16-bit NTTs with Plantard arithmetic

We now present the faster 16-bit NTT implementations using the Plantard arithmetic on Cortex-M3 and M4. The 16-bit NTTs we used in this paper include NTTs over $q_0 = 769$ and $q_1 = 3329$. Note that NTT over 769 is used on both platforms while NTT over 3329 is only deployed on Cortex-M3.

Plantard arithmetic for 16-bit modulus. The efficient implementation of Plantard multiplication by a constant for 16-bit modulus q_i on Cortex-M3 is shown in Algorithm 22 [65]. For its efficient implementation on Cortex-M4, we refer to [63, Algorithm 11]. In order to utilize the efficient Plantard arithmetic in our 16-bit NTT implementation, an important positive integer α_i that satisfies $q_i < 2^{16-\alpha_i-1}$ is required. For the moduli $q_0 = 769$ and $q_1 = 3329$ that we used in this paper, we set $\alpha_0 = 5$ for $q_0 = 769$ and $\alpha_1 = 3$ for $q_1 = 3329$. Besides, we also need to precompute $q_0' = q_0^{-1} \bmod 2^{32}$ and $q_1' = q_1^{-1} \bmod 2^{32}$ to carry out the Plantard arithmetic.

In 2023, Huang et al. [65] showed that the input range of the Plantard multiplication can be enlarged to $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. When b is a constant, we can always reduce it down to $[0, q_i)$; so when replacing the specific q_i and $b_{max} = q_i - 1$ into the input range $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$, we have $a \in [-2645q_0, 4541q_0]$ and $a \in [-137q_1, 230q_1]$ for $q_0 = 769$ and $q_1 = 3329$, respectively. When performing NTT on Cortex-M3, each coefficient loaded into the register could be bigger than

a 16-bit signed integer but we need to ensure that it is reduced down to a 16-bit signed integer when we need to store it back to the 16-bit memory [65]. However, the NTT on Cortex-M4 utilizes the SIMD instruction and the coefficients must fit in 16-bit signed integers. Hence, the safe coefficient range for a 16-bit signed integer is in $[-42q_0, 42q_0]$ and $[-9q_1, 9q_1]$ for q_0 and q_1, respectively.

NTTs over 769 and 3329. As the core operation of NTT, the efficiency of butterfly units is essential in NTT/INTT implementation. We adopt the state-of-the-art butterfly combination presented in [2, 3] for NTTs over 769 and 3329 on Cortex-M3 and M4, namely using Cooley-Turkey (CT) butterfly for both NTT and INTT. We follow the twiddle factor precomputation in [63, Section 4.2.1] in order to utilize the efficient Plantard multiplication by a constant for the modular multiplication by twiddle factor, namely $\zeta_i' = \zeta_i \cdot (-2^{32} \bmod q_i) \cdot (q_i^{-1} \bmod 2^{32}) \bmod 2^{32}$.

We follow the state-of-the-art 4+3 and 3+4 layer merging strategies in previous work [3, 63] for NTT and INTT on Cortex-M4. As for Cortex-M3, the 3+3+1 and 3+1+3 layer merging strategies for NTT and INTT on Cortex-M3 are adopted as in [65].

Range analysis. The input range of the 16-bit NTT must be smaller than the coefficient range of $\mathbf{t}_0$, i.e., 2^{12}. When using CT butterfly and Plantard arithmetic for NTT, each layer of NTT would increase the coefficient size by $q_i/2$; so seven layers of NTT would increase the coefficient size by $3.5q_i$. For both moduli 769 and 3329, NTT would generate coefficients smaller than $2^{12} + 3.5q_i < 2^{15}$, which can fit in 16-bit signed integers. Therefore, the NTT with Plantard arithmetic does not need any modular reduction. We will see that after the pointwise multiplication described in Section 4.5.3, the input range of INTT is in $[-(q_i + 1)/2, q_i/2)$.

On Cortex-M4, we are using INTT over 769 for $c\mathbf{s}_i$ and we need to ensure that every coefficient must not overflow a 16-bit signed integer. Since the INTT is implemented with CT butterfly similar to the implementation in [2, 3], it would bring

additional twisting in the last layer of INTT. To minimize the side-effect of the additional twisting, they [2, 3] introduced the light butterfly which could omit the same amount of modular multiplication. The first three layers and the first iteration of the last three layers of INTT are implemented with light butterflies. Since the input range of INTT is smaller $q_0/2$, the first 3-layer INTT with light butterfly increase some of the coefficients up to $4.5q_0$. The fourth layer of INTT with CT butterfly further increases these coefficients up to $5q_0$. Then, in the final three layers of INTT with light butterfly, these coefficients would be increased to $40.5q_0$, which can still fit in a 16-bit signed integer for q_0 (see Section 4.5.3). Therefore, the INTT over 769 can totally eliminate the modular reduction of coefficients. If we use NTT over 3329 for $c\mathbf{s}_i$, the INTT over 3329 needs modular reduction for 16 coefficients before the final three layers of INTT [63].

As for Cortex-M3, since the 16-bit coefficients are loaded into 32-bit registers, they can temporarily surpass 16-bit signed integers but need to be reduced down to 16-bit when they are stored back to 16-bit memory. Because we are using the 3+1+3 layer merging strategy, we just need to ensure that the coefficients can fit in 16-bit signed integers after the third and fourth layers of INTT. As mentioned above, the first four layers of INTT with CT butterfly produce coefficients smaller than $5q_i$, which can fit in 16-bit signed integers for both q_0 and q_1. And the final three layers of INTT with light butterfly would increase the coefficients up to $40.5q_i$. For $q_1 = 3329$, $40.5q_1$ surpass the 16-bit signed integer but fit in 32-bit signed integer, which is allowed on Cortex-M3. These coefficients will be reduced in the modular multiplication with the twiddle factor and 128^{-1} in the final layer of INTT. Therefore, we do not need any modular reduction in INTTs over 769 and 3329 on Cortex-M3.

The explicit CRT implementation with Plantard arithmetic on Cortex-M3. As shown in Algorithm 30, the explicit CRT is used to reconstruct the 32-bit results after INTT. The computation $u = u_0 + \big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big)\, q_0$ is performed for 256 pairs of coefficients (u_0 and u_1). During the computation, we can

see that $m_1 = q_0^{-1} \bmod^{\pm} q_1$ is a constant. Therefore, the modular multiplication with m_1 modulo q_1 could be optimized with the Plantard multiplication by a constant. The instruction sequence of the explicit CRT with Plantard arithmetic on Cortex-M3 is shown in Algorithm 31. To use the Plantard multiplication by a constant, we need to precompute the term $m_1' = m_1 \cdot (-2^{32} \bmod q_1) \cdot (q_1^{-1} \bmod 2^{32}) \bmod 2^{32}$ so that the Plantard arithmetic could produce result in normal domain. After Step 5 in Algorithm 31, the modular result t using the Plantard arithmetic is in range $[-(q_1+1)/2, q_1/2)$. The rest of the computation $u = u_0 + t \cdot q_0$ would be much smaller than the modulus q of Dilithium; so no further modular reductions are required. Overall, using the Plantard arithmetic in the explicit CRT computation could save one multiplication for each coefficient compared to the Montgomery-based implementation.

Algorithm 31 The explicit CRT with Plantard arithmetic on Cortex-M3

Input: $u_0 = u \bmod q_0, u_1 = u \bmod q_1, m_1 = q_0^{-1} \bmod^{\pm} q_1, m_1' = m_1 \cdot (-2^{32} \bmod q_1) \cdot (q_1^{-1} \bmod 2^{32}) \bmod 2^{32}, q_1 2^{\alpha_1} < 2^{15}$

Output: $u = u_0 + ((u_1 - u_0) m_1 \bmod^{\pm} q_1) q_0$

1: `sub` t, u_1, u_0

2: `mul` t, t, m_1'

3: `add` $t, 2^{\alpha_1}, t, \text{asr}\#16$

4: `mul` t, t, q_1

5: `asr` $t, t, \#16$ ▷ $t \leftarrow (u_1 - u_0) m_1 \bmod^{\pm} q_1$

6: `mla` u, t, q_0, u_0 ▷ $u \leftarrow u_0 + t q_0$

7: **return** u

Asymmetric multiplication for pointwise multiplication. The pointwise multiplication of incomplete NTTs over 769 and 3329 requires computing 128 degree-1 polynomial multiplications over $X^2 - \zeta_0^{2br_7(i)+1}$ and $X^2 - \zeta_1^{2br_7(i)+1}$ for $q_0 = 769$ and $q_1 = 3329$, respectively, with $i = 0, 1, \ldots, 127$ and $br_7(i)$ denotes the bit reversal of the 7-bit integer i. For simplicity, we denote the corresponding $\zeta_0^{2br_7(i)+1}$ and

$\zeta_1^{2br_7(i)+1}$ as ζ below. We adopt the asymmetric multiplication proposed in [2, 3] to speed up the degree-1 pointwise multiplication. For the polynomial multiplication ct, after the NTT transform, both c and t are transformed into NTT-domain in the form of $\hat{c}_{2i} + \hat{c}_{2i+1}X$ and $\hat{t}_{2i} + \hat{t}_{2i+1}X$. Then, the pointwise multiplication over $X^2 - \zeta$ is computed as $(\hat{c}_{2i} + \hat{c}_{2i+1}X) \circ (\hat{t}_{2i} + \hat{t}_{2i+1}X) = (\hat{c}_{2i}\hat{t}_{2i} + \hat{c}_{2i+1}\hat{t}_{2i+1}\zeta) + (\hat{c}_{2i}\hat{t}_{2i+1} + \hat{c}_{2i+1}\hat{t}_{2i})X$. Note that c is reused for $c\mathbf{s}_1$, $c\mathbf{s}_2$, $c\mathbf{t}_0$, and $c\mathbf{t}_1$, we can precompute the term $\hat{c}_{2i+1}\zeta$ with Plantard multiplication by a constant in advance to reduce repeated computations. Then, two Plantard reductions are used to perform $\hat{c}_{2i} \cdot \hat{t}_{2i} + \hat{c}_{2i+1} \cdot \hat{t}_{2i+1}\zeta \bmod q_i$ and $\hat{c}_{2i} \cdot \hat{t}_{2i+1} + \hat{c}_{2i+1} \cdot \hat{t}_{2i} \bmod q_i$, and all coefficients are reduced down to $[-(q_i + 1)/2, q_i/2)$.

4.6 Keccak optimizations on ARMv7-M

This section first reviews the existing optimization techniques and analyses the factors that affect the overall Keccak performance on ARMv7-M. Subsequently, two architecture-specific optimization techniques for enhancing Keccak on ARMv7-M are introduced.

4.6.1 Existing optimization techniques on ARMv7-M

The designers of Keccak have summarized many possible optimizations for software and hardware implementations in a dedicated document [29]. The two most useful ones on ARMv7-M are described below.

Bit-interleaving. The intuitive way to implement Keccak-p in software is to follow a lane-wise architecture (i.e. a register contains one or multiple lanes), as described in Listing 2.1. On platforms where registers are not large enough to handle an entire lane, the designers recommend using the bit-interleaving technique. For $b = 1600$ on 32-bit architectures, it consists of storing bits at odd positions in one register, and bits at even positions in another register. In this way, 64-bit rotations can be easily handled by separate 32-bit rotations without additional operations. Note that this

requires some extra calculations for rearranging the lanes at the beginning and at the end of the permutation. However it is calculated at a higher level (i.e. when adding and extracting data to and from the state), and therefore it is not taken into account when benchmarking the permutation itself.

Efficient in-place processing. When updating the internal state during the round function, it is possible to store all processed data back into the same memory location it was loaded from. In this way, only a single instance of the state (instead of two) must be preserved. Because the π operation moves lanes within the state, it requires defining a mapping between the lane coordinates and the memory location depending on the round number. The Keccak designers propose a linear map using a matrix of order 4, which means the state will return to its initial memory location after 4 rounds.

Performance analysis. On 32-bit architectures, each round of Keccak-p$[1600, \cdot]$ consists of 152 XORs, 50 ANDs, 50 NOTs and 58 rotations thanks to the bit-interleaving technique. On ARM, it is possible to merge 1 AND and 1 NOT into a single `bic` instruction and the rotations during the θ step can be easily combined with an XOR thanks to the inline barrel shifter. This leads to 250 instructions per round overall: 152 `eor`, 50 `bic` and 48 `ror`. Assuming logical instructions take 1 cycle, the raw cost of Keccak-p$[1600, 24]$ on this platform should theoretically be $250 \times 24 = 6\,000$ clock cycles. However, given the fact that the state is 1600-bit long and that ARMv7-M only offers 14 32-bit general-purpose registers to work with, it implies that performance will inevitably bear the cost of many loads and stores on the stack. As reference, we consider the ARMv7-M implementation from the extended Keccak Code Package (XKCP) [28]. Because it contains optimized, free and open-source implementations for various platforms and architectures, it is often used as a third-party software component. According to previous research, the assembly implementation of Keccak-p$[1600, 24]$ from XKCP requires 12 969 clock cycles on the Cortex-M4 [109], meaning that around 54% of the cycles are spent in memory

accesses. However, there is still room for improvement as explained hereafter.

```
1  .macro xor5   result,b,g,k,m,s
2      ldr       \result, [r0, #\b]
3      ldr       r1, [r0, #\g]
4      eors      \result, \result, r1
5      ldr       r1, [r0, #\k]
6      eors      \result, \result, r1
7      ldr       r1, [r0, #\m]
8      eors      \result, \result, r1
9      ldr       r1, [r0, #\s]
10     eors      \result, \result, r1
11 .endm
```

Listing 4.1: Original ARMv7-M assembly code from [28] to compute half a parity lane. Loads from memory are not fully grouped and thus not optimally pipelined on M3 and M4 processors.

```
1  .macro xor5  result,b,g,k,m,s
2      ldr      \result, [r0, #\b]
3      ldr      r1,  [r0, #\g]
4      ldr      r5,  [r0, #\k]
5      ldr      r11, [r0, #\m]
6      ldr      r12, [r0, #\s]
7      eors     \result, \result, r1
8      eors     \result, \result, r5
9      eors     \result, \result, r11
10     eors     \result, \result, r12
11 .endm
```

Listing 4.2: ARMv7-M assembly code after optimization to compute half a parity lane. Loads from memory are now fully grouped and thus optimally pipelined on M3 and M4 processors.

4.6.2 Pipelining memory accesses

The ARMv7-M assembly implementation provided by XKCP at the time of writing[5] works as follows. At the beginning of each round, all the parity lanes (namely `D[x]` on line 10 of Listing 2.1) are precomputed. To compute half a parity lane, the code relies on a macro `xor5` which consists of 5 `ldr` and 4 exclusive-OR (`eor`) instructions, as detailed in Listing 4.1. Once this precomputing phase is complete, it executes the θ, ρ, π, χ and ι steps by a group of 5 half lanes at a time.

On the Cortex-M3 and M4, it is clear that the `xor5` macro suffers pipeline stalls since only 2 out of the 5 `ldr` instructions are consecutive. While grouping all these loads together would allow saving 3 cycles per macro call, this requires to change the way variables are assigned to registers since `r1` is used as the only destination of `ldr` instructions in order to preserve the other registers for subsequent calculations. Nevertheless, we managed to relax the register pressure thanks to

[5] `https://github.com/XKCP/XKCP/commit/7fa59c0ec4b5802b7c269ddd9ef0ef35999b4f0f`

another register allocation, allowing us to pipeline all the 5 `ldr` together within the `xor5` macro as detailed in Listing 4.2. To further improve memory access pipelining, we also reordered some other instructions throughout the code. Notably, we moved `str` instructions after multiple `ldr` as much as possible. While these pipelining optimizations have been carried out manually, tools are being developed to achieve optimal assembly implementations in an automated manner as recently illustrated by the SLOTHY optimizer based on constraint solving [1].

4.6.3 Lazy rotations

The XKCP implementation makes use of explicit rotations for the ρ step through `ror` instructions. While it makes the code easy to follow, it requires 47 such instructions per round. As recently proposed by Becker and Kannwischer on AArch64 [24], one can omit those explicit rotations by means of *lazy rotations* (i.e. rotating the second operands thanks to the inline barrel shifter) during subsequent operations. They recommend to defer the explicit rotations until the θ step in the next round: once all the (unrotated) parity lanes have been calculated, then they are rotated explicitly. By proceeding this way, the (unrotated) state lanes are lazily rotated when treated as second operands during the XOR with the (rotated) parity lanes, so that the internal state is back to the classical representation and the process can be reiterated thereafter. While it would be also possible to keep deferring rotations, it would require to fully unroll the permutation code, resulting in substantial impacts on code size.

On AArch64, it leads to 3 explicit rotations instead of 5 since 2 deferred rotation values are in fact 0. While it should theoretically result in 6 explicit rotations on ARMv7-M because of its 32-bit architecture, it is possible to stick to 3 rotations overall as described below. Thanks to the bit-interleaving technique, computing a 1-bit rotation on a 64-bit word can be achieved using a single 32-bit rotation only. Therefore, when computing the parity lanes, only 5 `eor` instructions need to rotate their second operand, leaving the barrel shifter available for deferred rotations during

the remaining 5 `eor`. In the end, only 3 explicit rotations remain per round instead of 47.

We implemented this technique on ARMv7-M along with the in-place processing optimization in two different ways. While the most efficient approach is to use lazy rotations for all rounds, it requires to have specific routines for the first and last rounds to deal with input and output misalignments since the internal state is expected to be properly aligned at function entry and exit. When considering in-place processing, and therefore a quadruple round routine, it results in a code size increase by half since the first and last rounds should both come in two variants. In order to boost the performance while limiting the impact on code size, we also propose a variant where lazy rotations are applied for three-quarters of the rounds only. This way, explicit rotations are used every 4 rounds to ensure the internal state is correctly aligned when entering the quadruple round. Still, a potential drawback of deferring rotations is that it affects the code readability, which may make the integration of side-channel countermeasures (e.g. masking) more cumbersome.

4.7 Results and comparisons

4.7.1 Benchmark settings

For our benchmarks, we used the development boards and becnhmark settings described below. All implementations on on Cortex-M3 and M4 were compiled using `arm-none-eabi-gcc 10.2.1` along with the `-O3` optimization flag. We adopted the same configuration as the `pqm3`[6] and `pqm4` [71], and the clock cycles were measured using the `pqm3` and `pqm4` frameworks on Cortex-M3 and M4, respectively.

Cortex-M4 setup. For the Cortex-M4 platform, we utilize the STM32F407G-DISC1 and NUCLEO-L4R5ZI boards. The STM32F407G-DISC1 features a STM32F-407VG microcontroller running up to 168 MHz, 1024 KB of flash memory and 192 KB

[6] `https://github.com/mupq/pqm3`

of RAM divided into three blocks: two contiguous blocks of SRAM connected to the bus matrix with different interconnects, and a core coupled memory (CCM) block which is connected directly to the core. We exclusively used the 64 KB CCM to achieve the best performance with a clock core speed of 24 MHz to execute code from flash with zero-wait state. The NUCLEO-L4R5ZI board features 2MB of Flash and 640KB of RAM.

Cortex-M3 setup. For the Cortex-M3 platform, we utilize the ATSAM3X8E microcontroller on an Arduino Due board [44] as the testbed. Our implementation is based on the PQM3 repository [46], and the GCC v10.2.1 is used to compile the code. We leverage the hardware random number generators available on Cortex-M3 to obtain the necessary random numbers for the Kyber implementation. Stack usage is measured following the methodology outlined in [46]. Additionally, SHA-3 and SHAKE are implemented using the optimized assembly Keccak permutation token from the eXtended Keccak Code Package (XKCP)[7].

RISC-V setup. The realistic RISC-V platform we choose is the SiFive Freedom E310 board, which is equipped with a 32-bit E31 RISC-V core [106]. Due to its limited 16KiB memory, a direct deployment of Kyber on this platform is infeasible. Hence, in order to provide detailed comparisons with the existing Kyber implementation on RISC-V, we also present experimental results obtained from the RISC-V simulator used in PQRISCV [97], which uses a RISC-V CPU implemented in pqriscv-vexriscv[8] as the PQC benchmarking platform. The GCC compiler we used for RISC-V is the RISC-V GNU v10.2.0. Since both of the RISC-V platforms do not have hardware random number generators, we reuse the function in PQRISCV [97] to generate the random numbers in Kyber. Furthermore, for the implementation of SHA-3 and SHAKE, we rely on the optimized Keccak permutation provided by Ko [109] in PQRISCV.

[7] `https://github.com/XKCP/XKCP`

[8] https://github.com/mupq/pqriscv-vexriscv

To support the Kyber-90s variants, we adopt the optimized AES implementation presented in [4] on Cortex-M3 and PQRISCV similar to the PQM3 [46]. On the SiFive platform, however, we use the reference SHA-3, SHAKE, and AES implementations provided in SUPERCOP[9] to reduce the code size. This is because a bloated code size might impact the performance of the platform. Further discussion on this issue will be provided in Subsection 4.7.4.

4.7.2 Performance of Keccak

Table 4.1 provides the benchmark of Keccak-p$[1600, 24]$ for different implementation characteristics. The first one, denoted by `ldr/str`, refers to the memory access pipelining strategy. In this context, the term "mostly grouped" signifies that while the majority of memory accesses are packed together; this is not the case during the parity lanes precomputation. Another characteristic indicates the utilization and degree of lazy rotations. The pipeline optimizations achieved by our new register allocation result in significant performance improvements of 17.13 % and 12.84 % on Cortex-M3 and M4, respectively. Note that it comes at the cost of a slight code size increase due to the fact that some `ldr` instructions are now using the higher-half registers (i.e. `r8` to `r14`) as the destination, leading to a 32-bit encoding (versus 16-bit when using lower-half registers). When combined with the use of lazy rotations, we achieve up to 24.78 % and 21.4 % performance boosts on Cortex-M3 and M4, respectively. Nevertheless, the majority of the remaining instructions that were initially encoded on 16 bits have now been 32-bit encoded. This is because the second operands undergo systematic rotation using the inline barrel shifter. As discussed in Subsection 4.6.3, the variant that utilizes lazy rotations for all rounds incurs a 50% increase in code size for a minor performance gain of aroung 2%. While we could have potentially made it more compact, we believe that doing so would have compromised the performance advantage, resulting in no overall benefits compared to the other variant. Therefore, when code size is a critical factor, we

[9] `http://bench.cr.yp.to/supercop.html`

Table 4.1: Keccak-p[1600, 24] benchmark on Cortex-M3 and M4.

Ref.	Implementation characteristics*		Speed (clock cycles)		Code size	RAM
	ldr/str	lazy ror	M3	M4	(bytes)	(bytes)
XKCP	mostly grouped	✗	13 015	11 725	5 576	**264**
This work	grouped	✗	10 785	10 219	5 772	**264**
	grouped	✓ (3/4)	9 981	9 415	6 556	**264**
	grouped	✓ (4/4)	**9 789**	**9 218**	9 536	**264**

*All listed implementations take advantage of the in-place processing and bit-interleaving techniques.

suggest to favour the implementation that uses lazy rotations for three-quarters of the rounds only in order to minimize the memory footprint, which is also adopted in the following benchmark.

4.7.3 Speed performance of polynomial multiplications

NTT/INTT of Kyber and NTTRU on Cortex-M4 The cycle counts of the core polynomial arithmetic in Kyber and NTTRU are presented in Table 4.2. Using the improved Plantard arithmetic, the proposed Kyber implementation based on [12] achieves 20.24% and 16.92% speed-ups for NTT and INTT, respectively. The speed-optimized implementation of [3] trades the speed with extra stack usage while implementing the matrix-vector multiplication and base multiplication, which explains their two times faster base multiplication. Besides, the coefficients of their matrix-vector multiplication lie in $[-q, q]$, which reduces one modular reduction of 128 coefficients at the beginning of their INTT implementation. On the contrary, the matrix-vector multiplication of their stack-friendly implementation produces coefficients smaller than kq, which requires extra modular reductions in INTT and results in a slower INTT implementation.

After applying the improved Plantard arithmetic in their stack-friendly implementation, we halve the coefficient size of the matrix-vector multiplication down to $\frac{1}{2}kq$; then, three different lazy reduction strategies are adopted for the INTT in Kyber512, Kyber768, and Kyber1024, respectively. Overall, the proposed implemen-

Table 4.2: Cycle counts for the core polynomial arithmetic in Kyber and NTTRU on Cortex-M4.

Scheme	Implementation	NTT	INTT	Base Mult	Base Inv
Kyber	[12]	6 822	6 951	2 291	-
	This work[a]	5 441	5 775	2 421	-
	Speed-up	20.24%	16.92%	-5.67%	-
	Stack[3]	5 967	5 917	2 293	-
	Speed[3]	5 967	5 471	1 202	-
	This work[b]	4 474	4 684/4 819/4 854	2 422	-
	Speed-up (stack)	25.02%	20.84%/18.56%/17.97%	-5.58%	-
	Speed-up (speed)	25.02%	14.38%/11.92%/11.28%	-101.41%	-
NTTRU	[82]	102 881	97 986	44 703	100 249
	This work	17 274	20 931	10 550	40 763
	Speed-up	83.21%	78.64%	76.40%	59.34%

[a] Implementation based on [12], [b] Implementation based on the stack-friendly code of [3].

tation based on the stack-friendly code of [3] achieves 25.02% speed-up for NTT and 20.84%/18.56%/17.97% speed-ups for the INTT in Kyber512/Kyber768/Kyber1024, respectively. Our INTT implementation is also 14.38%/11.92%/11.28% faster than their speed-optimized INTT implementation in Kyber512/Kyber768/Kyber1024, respectively. All these speed-ups mainly come from the efficient Plantard multiplication by a constant and better lazy reduction strategies. The results clearly reveal the improved Plantard arithmetic's benefits to the NTT/INTT on Cortex-M4. The base multiplication of Kyber consists of modular multiplication by a constant and modular multiplication of two variables, which increases the register usage pressure and requires extra cycles to handle this problem.

The polynomial arithmetic implementation of NTTRU is similar to Kyber. By fully utilizing the SIMD instructions, the double butterflies, double modular reduction over packed coefficients, double base multiplication, and double base inversion are first presented by applying the improved Plantard arithmetic. The assembly implementation of NTTRU obtains excellent speed-ups compared with its reference

implementation in [82]. Table 4.2 shows that the speed-ups for NTT, INTT, base multiplication, and base inversion are 83.21%, 78.64%, 76.40%, and 59.34%, respectively. The speed-ups for NTT and INTT mainly come from the efficient Plantard multiplication by a constant, better lazy reduction techniques, and parallel implementation utilizing the SIMD instructions on Cortex-M4. The speed-up for base inversion is relatively smaller than others because most of the modular multiplications in this operation are implemented as the modular multiplications of two variables. Therefore, the efficient Plantard multiplication by a constant (Algorithm 19) can not be applied to base inversion.

NTT/INTT of Kyber on Cortex-M3 and RISC-V Table 4.3 presents the performance comparison of the polynomial arithmetic. The state-of-the-art Kyber implementation on Cortex-M3 was presented by Denisa et al. [57], which is also the latest implementation integrated in PQM3 [46]. On RISC-V, the closest related work also comes from Denisa et al. [56]. However, since the implementations on RISC-V [56] do not involve any memory optimization techniques, we are unable to deploy their Kyber implementation on the SiFive platform. Instead, we only incorporate their NTT and INTT implementations on the SiFive platform for comparison.

As mentioned in Section 4.4.2, we eliminate the modular reduction of coefficients for NTT, therefore, both the stack-version and speed-version NTT cost the same number of cycles. Specifically, the proposed NTT implementation obtains speedups of 26.19%, 34.76%, and 22.67% on the ARM Cortex-M3, SiFive Freedom E310, and PQRISCV, respectively, compared to the implementations in [57] and [56]. In terms of INTT, because we manage to eliminate the modular reduction of coefficients in the stack-version INTT of Kyber512 and the speed-version INTT of all three Kyber variants, the INTT of Kyber512 is slightly faster than the one in the stack-version INTT of Kybe768 and Kyber1024. Overall, our INTT achieve speedups of 32.57%/34.14%,55.53%/ 56.95%, and 43.36%/44.49% on the ARM Cortex-M3, SiFive Freedom E310, and PQRISCV, respectively, compared to the implementations in [57] and [56].

Table 4.3: Cycle counts for the core polynomial arithmetic in Kyber on Cortex-M3, SiFive Freedom E310, and PQRISCV.

Platform	Implementation	NTT	INTT	Base Multiplication
Cortex-M3	Denisa et al. [57]	10 874	13 049	4 821
	This work (stack)	8 026	8 594/8 799	4 311
	This work (speed)	8 026	8 594	3 028/3 922/5 851
	Speedup (stack)	26.19%	34.14%/32.57%	1.06%
	Speedup (speed)	26.19%	34.14%	37.19%/18.65%/-21.37%
SiFive Freedom E310	Denisa et al. [56]	24 353	36 513	-[a]
	This work (stack)	15 888	15 719/16 227	10 020
	This work (speed)	15 888	15 719	4 893/5 662/9 313
	Speedup (stack)	34.76%	56.95%/55.53%	-
	Speedup (speed)	34.76%	56.95%	-
PQRISCV	Denisa et al. [56]	28 417	42 636	-[a]
	This work (stack)	21 975	23 666/24 146	12 236
	This work (speed)	21 975	23 666	7 747/9 795/13 068
	Speedup (stack)	22.67%	44.49%/43.37%	-
	Speedup (speed)	22.67%	44.49%	-

a. [56] did not provide results for base multiplication.

Huang et al. [63] reported that the stack-version base multiplication with the Plantard arithmetic is slower than the previous Montgomery-based counterpart on Cortex-M4 due to the lack of register usage resulting from the Plantard multiplication of two variables. However, on Cortex-M3 and RISC-V platforms, the stack-version base multiplication does not have this issue thus could even benefit from the Plantard multiplication by the twiddle factor. The speed-version implementation offers three variants of base multiplication to deploy the asymmetric multiplication and better accumulation techniques described in Section 4.5.3, which help to mitigate the need of some modular reductions. Therefore, two of the three variants are 29.74% and 9.02% faster than our stack-version base multiplication, while the third variant is 35.72% slower, mainly due to additional loading cycles and high register pressure. Compared to the state-of-the-art Montgomery-based implementation in

in [57], these two variants of the speed-version base multiplications are 37.19% and 18.65% faster than the base multiplication due to the use of the improved Plantard arithmetic, asymmetric multiplication and better accumulation techniques. Since the implementation from Denisa et al. [56] did not provide results for the base multiplication, we do not compare it with our results on the RISC-V platforms. Overall, the speed-version implementation offers faster INTT and base multiplications for all variants of Kyber than the stack-version. It should be noted that the slower variant of base multiplication will only be used at the final stage of the innver-vector product and matrix-vector product while the two faster variants will be used repeatedly in these two operations. Therefore, the faster INTT and base multiplications would speed up these two time-consuming operations. Together with the proposed memory optimizations in Section 4.4.3, the speed-version implementation provides an effective time-memory trade-off for Kyber on low-end 32-bit platforms. Detailed improvements of the speed-version implementation could be found in Table 4.7.

NTT/INTT of Dilithium on Cortex-M3 and M4 Table 4.4 presents results for NTT-related functions on Cortex-M3 and M4. Due to the instruction limits of Cortex-M3, the 16-bit NTT on Cortex-M3 is significantly faster than both constant-time and variable-time 32-bit NTT. To be more specific, the 16-bit NTT, INTT, and pointwise multiplication are 4.22×, 4.29×, and 2.14× faster than the constant-time 32-bit NTT, INTT, and pointwise multiplication, respectively. Compared to the 32-bit variable-time NTT, INTT, and pointwise multiplication, the speed-ups are 2.48×, 2.46×, and 1.24×, respectively.

The highly efficient 16-bit NTT implementations makes the proposed multi-moduli NTT practical on Cortex-M3. Compared to the constant-time 32-bit NTT, the proposed multi-moduli NTT, INTT and pointwise multiplication implementations yield 52.76% $\sim$ 54.76% performance improvements. Besides, the multi-moduli NTT and INTT also obtain 19.47% and 19.07% speed-ups compared with the variable-time 32-bit NTT and INTT. The pointwise multiplication is 63.05% slower than theirs but we will later show that our multi-moduli NTT implemen-

Table 4.4: Cycle counts for the NTT-related functions in Dilithium on Cortex-M3 and M4. Averaged over 1000 executions.

Platform	Prime	Ref.	NTT	INTT	Pointwise	CRT
M3	8380417	[57] constant-time	33 077	36 661	8 528	✗
	8380417	[57] variable-time	19 405	21 051	4 944	✗
	3329×7681	[2] (Saber)	16 770	19 056	11 927	4 637
	769	This work	7 830	8 543	3 989	✗
	769×3329	This work	15 626	17 037	8 061	3 735
M4	8380417	[3]	8 066	8 388	1 931	✗
	257	[3]	5 497	5 540	1 201	✗
	769	[3]	5 180	5 512	1 715	✗
	769	This work	4 456	4 593	1 717	✗

tation for small polynomial multiplications is still better than the one that uses variable-time 32-bit NTT. We also compare our multi-moduli NTT implementation with a similar implementation for Saber in [2]. Although [2] adopted a faster 6-layer NTT/INTT, our 7-layer NTT/INTT implementation using the efficient Plantard arithmetic is still slightly faster, obtaining 6.82%, 10.60%, and 32.41% speed-ups for NTT, INTT, and pointwise multiplication, respectively. Moreover, since the explicit CRT can also be optimized with the efficient Plantard multiplication by a constant, our CRT implementation is also 19.45% faster. In summary, using Plantard arithmetic in NTT, INTT and CRT makes the multi-moduli NTT more efficient on Cortex-M3.

As for Cortex-M4, we integrate the efficient Plantard arithmetic for the NTT over 769. As shown in Table 4.4, our NTT and INTT implementations achieves $13.98\% \sim 18.94\%$ speed-ups compare with FNT over 257 and NTT over 769 in [3], respectively. The speed-ups of 16-bit NTT and INTT versus 32-bit NTT and INTT are 44.76% and 45.24%, respectively. Because the proposed multi-moduli NTT implementation requires to perform double 16-bit NTTs, INTTs, and pointwise multiplications, the 16-bit NTT/INTT needs to be at least 50% faster than the

Table 4.5: Cycle counts for small polynomial multiplications in Dilithium on Cortex-M3 and M4. Averaged over 1000 executions.

Platform	Operation	Dilithium2		Dilithium3		Dilithium5	
		[57]	This work	[57]	This work	[57]	This work
M3	$c\mathbf{s}_1$	$346k$	$106k$	$424k$	$128k$	$580k$	$172k$
	$c\mathbf{s}_2$	$346k$	$106k$	$502k$	$150k$	$658k$	$194k$
	$c\mathbf{t}_0$	$269k$	$195k$	$328k$	$284k$	$446k$	$372k$
	$c\mathbf{t}_1$	$213k$	$195k$	$311k$	$284k$	$409k$	$372k$
		[3]	This work	[3]	This work	[3]	This work
M4	$c\mathbf{s}_1$	$56k$	–	$79k$	$70k$	$92k$	–
	$c\mathbf{s}_2$	$56k$	–	$89k$	$78k$	$110k$	–

32-bit NTT/INTT in order to make it faster than the 32-bit NTT. Therefore, the multi-moduli NTT is not applicable on Cortex-M4.

Speed of $c\mathbf{s}_i$ and $c\mathbf{t}_i$ in Dilithium on Cortex-M3 and M4. Table 4.5 compares the performance of small polynomial multiplications $c\mathbf{s}_i$ and $c\mathbf{t}_i$ on Cortex-M3 and M4. The secret vector $\mathbf{s}_1$ is l-dimensional vector while the other three vectors ($\mathbf{s}_2, \mathbf{t}_0$, and $\mathbf{t}_1$) are all k-dimensional vectors. On Cortex-M3, there are three different implementation strategies based on whether the operations are secret-related operations. The $c\mathbf{s}_i$ is secret-related and previous work [57] implements it with constant-time 32-bit NTT on Cortex-M3. Compared to their implementation with constant-time 32-bit NTT, our $c\mathbf{s}_i$ implementation with 16-bit NTT is not only constant-time but also $3.26\times \sim 3.39\times$ faster. Since Dilithium's security does not rely on $\mathbf{t}_i$ being secret, the $c\mathbf{t}_0$ implementation in [57] is not totally constant-time. They implement both c and $\mathbf{t}_0$ with 32-bit constant-time NTT, while the pointwise multiplication and INTT are implemented in variable-time. Compared to their $c\mathbf{t}_0$ implementation, our multi-moduli NTT implementation is constant-time and shows 27.51%, 13.41%, and 16.59% speed-ups for three variants of Dilithium, respectively. Besides, despite the $c\mathbf{t}_1$ implementation in `verify` in [57] is totally variable-time, our

Table 4.6: Cycle counts (cc) and stack usage (Bytes) for KeyGen, Encaps, and Decaps in Kyber and NTTRU on Cortex-M4.

Scheme	Implementation	KeyGen			Encaps			Decaps		
		$k = 2^a$	$k = 3$	$k = 4$	$k = 2$	$k = 3$	$k = 4$	$k = 2$	$k = 3$	$k = 4$
Kyber	[12]	454k	741k	1 177k	548k	893k	1 367k	506k	832k	1 287k
		2 464	2 696	3 584	2 168	2 640	3 208	2 184	2 656	3 224
	This work[b]	446k	729k	1 162k	542k	885k	1 357k	497k	818k	1 270k
		2 464	2 696	3 584	2 168	2 640	3 208	2 184	2 656	3 224
	Stack[3]	439k	717k	1 139k	534k	871k	1 329k	484k	797k	1 233k
		2 608	3 056	3 576	2 160	2 660	3 236	2 176	2 676	3 252
	Speed[3]	438k	711k	1 129k	531k	864k	1 316k	479k	787k	1 217k
		4 268	6 732	7 748	5 252	6 284	7 292	5 260	6 308	7 300
	This work[c]	430k	702k	1 119k	526k	861k	1 314k	472k	780k	1 211k
		2 608	3 056	3 576	2 160	2 660	3 236	2 176	2 676	3 252
NTTRU	[82]	526k			431k			559k		
		9 384			8 748			10 324		
	This work	267k			237k			254k		
		9 372			7 452			8 816		

[a] k is the dimension of the underlying Module-LWE problem for Kyber.

[b] Implementation based on [12].

[c] Implementation based on the stack-friendly code of [3].

multi-moduli NTT implementation still obtains 8.45%, 8.68%, and 9.05% speed-ups for three variants of Dilithium, respectively. However, because using multi-moduli NTT in `verify` would introduce an extra INTT for a k-dimensional polynomial vector, we did not apply it to `verify`. As for Cortex-M4, by utilizing the efficient Plantard arithmetic on the NTT over 769 for Dilithium3, we yield 11.39% and 12.36% speed-ups for $c\mathbf{s}_1$ and $c\mathbf{s}_2$ on Cortex-M4. Since we reuse the FNT over 257 for Dilithium2 and Dilithium5, we do not provide comparison of $c\mathbf{s}_i$ for them.

4.7.4 Performance of schemes

Note that the first row of each entry in Table 4.6 to Table 4.8 indicates the cycle count and the second row refers to stack usage.

Performance of Kyber and NTTRU on Cortex-M4. Note that these results do not include the Keccak optimizations presented in Section 4.6. Table 4.6 shows the cycle counts and stack usage of the KEM protocols, namely key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps). The Kyber implementation based on either [12] or the stack-friendly code of [3] achieves better speed performance than its Montgomery-based counterpart with the same stack usage. For the Kyber implementation based on [12], the speed-ups for KeyGen, Encaps, and Decaps are 1.27%-1.76%, 0.73%-1.09%, and 1.32%-1.78%, respectively. As for the Kyber implementation based on the stack-friendly code in [3], we achieve speed-ups of 1.76%-2.09%, 1.13%-1.50%, and 1.78%-2.48% for KeyGen, Encaps, and Decaps, respectively. It is worth noting that our implementation based on the stack-friendly version of [3] outperforms their high-speed version with approximately half of the stack. If the stack usage is not one of the primary improved goals, one can also integrate the improved Plantard arithmetic into their speed-optimized implementation for better efficiency. As stated in previous work [71], the dominance of the hashing functions of Kyber will reduce the overall effect of the optimized polynomial arithmetic, which explains the relatively small speed-ups for the KEM protocols of Kyber.

As the first assembly NTTRU implementation on Cortex-M4, we obtain 49.24%, 45.01%, and 54.56% speed-ups for KeyGen, Encaps, and Decaps compared with its reference implementation. The excellent speed-ups mainly come from the highly optimized assembly implementation of NTT, INTT, base multiplication, and base inversion implemented with the improved Plantard arithmetic. It is obvious that NTTRU outperforms every variant of Kyber by a large margin in terms of speed on Cortex-M4. Compared with the fastest Kyber512, NTTRU provides 37.91%, 55.03%, 46.19% faster KeyGen, Encaps, and Decaps with approximately 3.45$\sim$4.05 times larger stack usage, respectively. In sum, although it is not a candidate for the NIST PQC competition, its efficiency might make it suitable for some specific application scenarios or platforms.

Table 4.7: Cycle counts (cc) and stack usage (Bytes) of KeyGen, Encaps, and Decaps of Kyber on Cortex-M3, SiFive Freedom E310, and PQRISCV.

Platform	Implementation	Kyber512			Kyber768			Kyber1024		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Cortex-M3	Denisa et al.[57]	541k	650k	622k	878k	1 054k	1 010k	1 388k	1 602k	1 543k
		2 212	2 300	2 308	3 084	2 772	2 788	3 596	3 284	3 300
	This work (stack)	519k	628k	590k	844k	1 025k	967k	1 342k	1 563k	1 486k
		2 212	2 300	2 308	3 084	2 772	2 788	3 596	3 284	3 300
	This work (speed)	518k	626k	587k	842k	1 017k	958k	1 333k	1 548k	1 471k
		3 268	3 860	3 860	4 044	4 636	4 636	4 812	5 404	5 404
PQRISCV	Denisa et al.[56]	2 229k	2 927k	2 856k	4 166k	5 071k	4 957k	6 696k	7 809k	7 662k
		6 544	9 200	9 984	10 640	13 808	14 944	15 760	19 440	21 056
	This work (stack)	1 937k	2 355k	2 100k	3 147k	3 822k	3 467k	4 964k	5 794k	5 344k
		2 408	2 488	2 520	2 952	3 016	3 032	3 464	3 528	3 544
	This work (speed)	1 926k	2 339k	2 084k	3 104k	3 768k	3 413k	4 890k	5 704k	5 254k
		3 432	4 024	4 040	4 216	4 808	4 840	5 032	5 608	5 656
SiFive Freedom E310	This work (stack)	1 497k	1 812k	1 601k	2 413k	2 929k	2 635k	3 794k	4 435k	4 045k
		2 580	2 660	2 708	3 060	3 124	3 156	3 572	3 636	3 668
	This work (speed)	1 597k	1 903k	1 674k	2 731k	3 203k	2 919k	-	-	-
		3 620	4 212	4 244	4 340	4 932	4 964	-	-	-

Performance of Kyber on Cortex-M3 and RISC-V. Note that these results do not include the Keccak optimizations presented in Section 4.6. Table 4.7 and Table 4.8 presents the cycle counts of the KEM protocols for Kyber and Kyber-90s, respectively, including key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps), which are obtained by repeating each protocol one hundred times and then computing the average results. The stack usage is measured in a similar way to [46] and [56]. We have implemented the stack-version and speed-version of Kyber on Cortex-M3 and RISC-V platforms similar to the optimized implementation on Cortex-M4 [3]. Our stack-version implementation on Cortex-M3 offers speedups of 3.38%∼5.14%, 2.75%∼4.26%, and 2.43%∼3.69% compared to [57] for Kyber512, Kyber768, and Kyber1024, respectively, while having the same stack usage as theirs. Similarly, our speed-version implementation on Cortex-M3 outperforms the work in [57] with a speedup of 3.69%∼5.63%, 3.51%∼5.15%, and 3.37%∼4.67% for Kyber512, Kyber768, and Kyber1024, respectively, with 1.31∼1.68 times stack usage compared to their implementation. Notably, thanks to the proposed memory

Table 4.8: Cycle counts (cc) and stack usage (Bytes) for KeyGen, Encaps, and Decaps in Kyber-90s on Cortex-M3, SiFive Freedom E310, and PQRISCV.

Platform	Implementation	Kyber512-90s			Kyber768-90s			Kyber1024-90s		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Cortex-M3	Denisa et al.[57]	467k	527k	558k	782k	868k	909k	1 224k	1 332k	1 383k
		2 904	2 992	3 000	3 432	3 504	3 512	4 636	4 000	4 016
	This work (stack)	445k	505k	526k	748k	839k	865k	1 178k	1 293k	1 326k
		2 904	2 992	3 000	3 432	3 504	3 512	4 636	4 000	4 016
	This work (speed)	443k	502k	523k	741k	828k	854k	1 170k	1 274k	1 306k
		4 000	4 592	4 592	4 776	5 368	5 368	5 544	6 136	6 136
PQRISCV	Denisa et al.[56]	3 042k	3 491k	3 627k	6 106k	6 677k	6 854k	10 246k	10 953k	11 181k
		6 656	9 312	10 096	10 752	13 920	15 056	15 872	19 552	21 168
	This work (stack)	2 667k	2 902k	2 854k	5,126k	5,434k	5,370k	8 578k	8 983k	8 908k
		2 704	2 616	2 648	3 248	3 144	3 160	3 760	3 656	3 672
	This work (speed)	2 654k	2 836k	2 788k	5 081k	5 379k	5 314k	8,499k	8,888k	8,813k
		3 744	4 216	4 232	4 528	5 000	5 032	5 296	5 752	5 800
SiFive Freedom E310	This work (stack)	8 326k	8 619k	8 884k	11 443k	11 758k	12 061k	-	-	-
		2 780	2 860	2 908	3 292	3 356	3 388	-	-	-
	This work (speed)	8 438k	8 757k	9 046k	-	-	-	-	-	-
		3 820	4 412	4 444	-	-	-	-	-	-

optimizations in Section 4.4.3, our speed-version Kyber implementation on Cortex-M3 reduces the stack usage by 23.50%∼28.31% compared to its counterpart on Cortex-M4 [3, Table 4].

To the best of our knowledge, there exists only one Kyber implementation on RISC-V (cf. Denisa et al. [56]). They have provided optimized assembly implementation for NTT/INTT of Kyber and leave rooms for further exploration to optimize memory footprints using memory optimized techniques. Table 4.7 shows that their implementation has a stack usage that is 2.71∼5.94 times larger than our stack-version implementation on PQRISCV. Notably, our stack-version implementation achieves a speedup of 13.10%∼26.47%, 24.46%∼30.05%, and 25.80%∼30.25% for the three variants of Kyber, respectively, compared to their implementation. Moreover, our speed-version implementation outperforms theirs by 13.59%∼27.03%, 25.49%∼31.15%, and 26.96%∼31.43% using only 26.86%∼52.44% of their stack usage for the three variants of Kyber, respectively. It can be deduced that the large speedups of our implementations compared to [56] mainly stems from the fact that their implementation is not as optimized and does not incorporate the recent opti-

mization strategies described in the recent literature [32, 12, 3].

The extremely large stack usage of Denisa et al.'s implementation [56] makes it infeasible to deploy on the 16KiB SiFive board. Hence, we only provide results for the proposed optimized implementations on this platform. Our results demonstrate that with the memory optimization strategies proposed in Section 4.4.3, it is possible to deploy the speed-version of Kyber512 and Kyber768 on the selected SiFive board. Notably, these two variants of Kyber are unable to be deployed on this platform before using our memory optimizations. However, since the speed-version implementation has a larger code size than the stack-version, for example, 13.75KiB versus 12.44KiB for Kyber512, it is slower than the stack-version on the SiFive board. This can be attributed to the fact that the larger code size of the speed-version implementation spills more instructions into the ROM, leading to much slower performance compared to the stack-version, as stated in [115, Section 5.3]. However, as demonstrated in our Cortex-M3 and PQRISCV results, the speed-version implementation can still outperform the stack-version on different platforms with different characteristics. Although the performance of the speed-version implementation on the SiFive board is slower than the stack-version, we show the feasibility of deploying both stack-version and speed-version Kyber implementations on memory-constrained IoT devices, except for the speed-version of Kyber1024. This is made possible by the memory optimized strategies proposed in Section 4.4.3.

As for the Kyber-90s variants, we yeild 2.93% $\sim$ 6.27% speed-ups on Cortex-M3 and 12.33% $\sim$ 23.13% speed-ups on PQRISCV. The speed improvements are very similar to the Kyber variants on Cortex-M3 and PQRISCV. On the SiFive platform, the adoption of the reference AES implementation results in the Kyber-90s variants being over 3$\times$ slower than the Kyber variants. We did not utilize the optimized AES implementation in [4] because it would increase the code size, leading to slower performance. Additionally, the reference AES implementation introduces additional stack usage, posing challenges on the deployment of the speed-version of Kyber768-90s and both versions of Kyber1024-90s on the selected memory-constrained SiFive

Table 4.9: Cycle counts of Dilithium on Cortex-M3. Averaged over 1000 executions.

Operation	Dilithium2		Dilithium3		Dilithium5	
	[57]	This work	[57]	This work	[57]	This work
`keygen`	2 059k	1 739k	3 594k	3 011k	✗	5 034k
`sign`	7 139k	5 582k	11 916k	9 087k	✗	20 193k
`verify`	1 949k	1 648k	3 283k	2 755k	✗	4 694k

platforms. Therefore, we are unable to provide their results on the SiFive platform. Addressing these deployment issues for Kyber-90s will require additional efforts and optimizations. However, we believe that if the target platforms have AES hardware support, the Kyber-90s variants will remain a promising option on these platforms.

Performance of Dilithium on Cortex-M3. Combining the optimizations of Keccak and polynomial multiplication, our Dilithium implementation shows significant improvements compared with the state-of-the-art implementation on Cortex-M3. As shown in Table 4.9, we yield 15.54% and 16.22% speed-ups for the `keygen` of Dilithium2 and Dilithium3, respectively. The speed-ups for the `verify` of Dilithium2 and Dilithium3 are 15.44% and 16.08%, respectively. The speed-ups for `sign` are larger than `keygen` and `verify` thanks to our polynomial multiplication optimizations, ranging from 21.81% and 23.74% for Dilithium2 and Dilithium3, respectively. Since Dilithium5 consumes a large stack usage and it cannot be directly deployed in Cortex-M3, we adopt a basic on-the-fly matrix generation strategy, which is similar to streaming $\mathbf{A}$ in [57, Section 5.3], to avoid storing the whole matrix; thus making Dilithium5 deployable on Cortex-M3. Since both pqm3 and [57] did not provide Dilithium5 implementation on Cortex-M3, we could not conduct any comparison.

Performance and hash profiling of Kyber and Dilithium on Cortex-M4. Because Keccak is extensively used by the PQC algorithms selected by NIST for

Table 4.10: Cycle counts and hash profiling of Kyber and Dilithium on the Cortex-M4 using the pqm4 framework. Averaged over 1000 executions.

Scheme	Keccak Impl.	keygen		sign/encaps		verify/decaps	
		speed	hashing	speed	hashing	speed	hashing
Dilithium2	XKCP	1 595k	83.47%	4 052k	64.53%	1 576k	80.47%
	This work	1 357k	80.57%	3 487k	60.02%	1 350k	77.2%
Dilithium3	XKCP	2 828k	85.54%	6 523k	62.95%	2 702k	82.62%
	This work	2 394k	82.92%	5 574k	58.97%	2 302k	79.61%
Dilithium5	XKCP	4 817k	86.6%	8 534k	68.08%	4 714k	84.69%
	This work	4 069k	84.14%	7 730k	63.05%	3 998k	81.95%
Kyber512	XKCP	432k	80.12%	527k	82.86%	472k	73.76%
	This work	369k	76.75%	448k	79.85%	409k	69.74%
Kyber768	XKCP	704k	79.04%	860k	82.38%	778k	74.75%
	This work	604k	75.59%	732k	79.32%	674k	70.84%
Kyber1024	XKCP	1 122k	79.58%	1 314k	82.46%	1 208k	76.07%
	This work	962k	76.18%	1 119k	79.41%	1 043k	72.29%

standardization, our Keccak optimizations can also improve their performance on ARMv7-M. To highlight the improvements brought by our Keccak optimizations, we provide benchmarks and hash profiling based on the pqm4[71] for both Dilithium and Kyber on Cortex-M4. As shown in Table 4.10, for most of the Dilithium variants, the Keccak optimizations yield speed-ups ranging from 13.94% to 15.52%. However, because the Keccak permutation is not the primary operation in the rejection sampling process, the speed-up on the `sign` algorithm of Dilithium5 is relatively small, around 9.42%. Similarly, the speed-version Kyber KEM protocols in [71, 63] also experience speed-ups of 13.35% to 15.00% due to our Keccak optimizations. Additionally, we present the hash profiling results for these two schemes to showcase the efficiency of our Keccak optimizations. As can be seen from Table 4.10, our Keccak optimizations decrease the proportion of time spent on hashing by 2.46% $\sim$ 5.03%.

4.8 Security and extensibility

In order to achieve the security objective as described in Section 1.3, our implementation avoids the use of non-constant-time instructions on the target IoT devices. Additionally, we ensure that all the secret-related operations in our implementation are constant-time, and our implementation does not leak more sensitive information compared to the reference work, making it secure against simple power analysis and timing side-channel attacks.

In terms of extensibility, the proposed Plantard arithmetic, as detailed in Algorithm 16 and Algorithm 17, is not restricted to any specific modulus. It can be customized for various odd moduli that meet the prerequisites outlined in Theorem 3.4.1. Depending on the instruction set architecture (ISA) properties of the target platforms, the proposed Plantard arithmetic can be applied to other LBC schemes with 16-bit/32-bit odd moduli on 32-bit/64-bit platforms, as demonstrated in Subsection 4.3.3. This flexibility enables it to replace Montgomery and Barrett arithmetic, thereby accelerating LBC operations on the target platforms.

Regarding the optimization techniques for Keccak, while they are closely tied to the ARMv7-M architecture, they hold significant potential for enhancing the performance of any PQC schemes where hashing operations are a key performance bottleneck. Additionally, many of the optimizations for NTT/INTT can be adapted to other LBC schemes with 16-bit NTTs. However, it is worth noting that the lazy reduction strategy may require redesign in certain cases, as the Plantard arithmetic may necessitate different input ranges depending on the specific moduli used in other LBC schemes. The memory optimizations proposed for the speed-version Kyber implementation can also be extended to other low-end IoT platforms with similar memory constraints, allowing for more efficient use of limited resources. Lastly, the multi-moduli NTT approach and the use of small 16-bit NTTs in Dilithium can be adapted to other platforms like AVX2. Notably, the 16-bit NTT implementation[10]

[10] `https://github.com/pq-crystals/kyber`

on AVX2 is approximately 4.11× faster than the 32-bit NTT[11] on the same platform, which demonstrates its potential to apply the multi-moduli NTT and further boost the performance of Dilithium.

4.9 Conclusion

This chapter presented the optimized implementations of modular arithmetic, 16-bit NTT in Kyber and NTRU, small NTT and multi-moduli NTT in Dilithium, as well as the Keccak on Cortex-M4, Cortex-M3, and RISC-V platforms. These optimizations significantly enhanced the performance of LBC schemes on IoT devices. Additionally, the memory optimizations for the speed-version Kyber were discussed, aimed at improving its feasibility and practicality on memory-constrained IoT devices.

[11] https://github.com/pq-crystals/dilithium

Chapter 5

Efficient Masking-friendly Lattice-based Cryptography

The previous chapters explored efficient LBC implementations without incorporating tailored countermeasures to address side-channel attacks. The algorithmic design of these LBC schemes did not prioritize side-channel resistance, leaving them vulnerable to such threats [73, 83, 48]. Masking is a promising countermeasure to protect LBC schemes against side-channel attacks. However, applying masking to these schemes often incurs significant overhead due to the requirement of costly mask conversions [55, 39, 38, 20, 85, 95]. To address this challenge, Raccoon is specifically designed to be masking-friendly by exclusively utilizing arithmetic masking, thereby eliminating the need for expensive mask conversions.

This chapter presents the efficient implementation of the masking-friendly Raccoon, offering robust and efficient countermeasures to secure against side-channel attacks.

5.1 Introduction

Raccoon is a lattice-based digital signature scheme designed with a focus on being masking-friendly. Similar to Dilithium, its security is based on the hardness of the MLWE and MSIS problems. Although Raccoon was submitted to the NIST

Call for Additional Digital Signature Schemes and did not advance to the second round, its masking-friendly characteristics make it a promising candidate for specific applications, including smart cards, hardware security modules, and IoT platform security.

In recent years, several side-channel attacks have been proposed against Dilithium [72, 49, 110] and Falcon [74, 58, 117]. The primary countermeasure against such attacks is masking [69]. Unfortunately, lattice-based signature schemes like Dilithium and Falcon include subroutines where masking is computationally expensive. For instance, in the masked implementation of Dilithium-Sign, the most efficient method for sampling random errors involves sampling in a Boolean masked form followed by a conversion to an arithmetic masked form. This process requires expensive mask conversions [55, 39, 38]. Although the efficiency of mask conversions has improved over time, securely implementing these conversions still incurs a computational cost of at least $O(d^2)$, where d represents the number of shares the sensitive data is split into. Moreover, rejection sampling, another critical component, also relies heavily on expensive mask conversions [20, 21, 85, 52]. In contrast, the masked implementation of Raccoon avoids the need for mask conversions and achieves a lower computational overhead of $O(d \log d)$.

This work is motivated by the following factors: (1) Currently, there is no assembly-optimized implementation of Raccoon for the ARM Cortex-M4 platform. Moreover, the reference implementation of Raccoon relies on the `__int128` data type, which is incompatible with 32-bit platforms like the Cortex-M4. (2) High-order masked implementations of Raccoon consume significant memory, making it challenging to deploy on memory-constrained IoT devices. Therefore, this work explores memory optimization techniques to facilitate the deployment of masked Raccoon implementations on such devices.

Contributions. Our contributions is shown as follow:

- In Raccoon's multi-moduli NTT implementation, each 64-bit coefficient requires double modular reductions. We reexamine two variants of Montgomery

reduction for this operation and observe that the state-of-the-art 32-bit Montgomery reduction on the Cortex-M4 is not efficiently implementable in-place for double modular reductions. In contrast, Seiler's variant of Montgomery reduction allows for a more efficient in-place implementation. However, the `smmls` instruction on the Cortex-M4 is incompatible with the computational process in Seiler's Montgomery reduction. To address this issue, we propose a novel approach: negative double Montgomery reductions. This technique reduces the instruction count for double modular reductions to just five. While this approach produces negative results, as shown in Equation 5.3.3, it does not impact subsequent computations, as demonstrated in Section 5.3.1.

- Additionally, we show that the lazy reduction strategy is not only applicable in the NTT/INTT implementation but also improves the efficiency of four key subroutines in Raccoon. We perform a detailed range analysis to ensure the correctness of the Raccoon implementation with the lazy reduction technique. Additionally, we optimize several 64-bit polynomial arithmetic operations tailored to Raccoon's parameters, such as left-shift/right-shift, coefficient reduction, and conditional operations.

- Finally, we explore memory optimization techniques for the lightweight implementations of the speed-optimized Kyber and high-order Raccoon, addressing the challenges posed by memory-constrained IoT devices. These optimizations significantly improve the feasibility and practicality of deploying the speed-optimized Kyber and high-order Raccoon, ensuring robust quantum-safe protection and side-channel resistance on memory-limited devices.

5.2 Target platforms

This chapter also targets the Cortex-M4 platform, which is one of the IoT platforms recommended by NIST for evaluating the performance of PQC schemes. A detailed introduction to the Cortex-M4 platform can be found in Section 4.2, and only a few

of the instructions used in this chapter will be introduced here.

We represent the product of two 32-bit integers as $a \cdot rm$, the lower 32 bits of this product as $\text{low}_{32}(rn \cdot rm)$, and the higher 32 bits as $\text{high}_{32}(rn \cdot rm)$. A 64-bit integer $rd_l + rd_h \cdot 2^{32}$ is expressed as $\{rd_l, rd_h\}$. The following are several commonly used instructions, with those beginning with "s" indicating signed integer operations.

The instruction `mul` rd, rn, rm computes $rd \leftarrow \text{low}_{32}(rn \cdot rm)$. The instruction `smlal` rd_l, rd_h, rn, rm calculates $\{rd_l, rd_h\} \leftarrow \{rd_l, rd_h\} + rn \cdot rm$. The `smmul` rd, rn, rm instruction computes $rd \leftarrow \text{high}_{32}(rn \cdot rm)$. The `smmla` rd, rn, rm, ra instruction computes $rd \leftarrow \text{high}_{32}(rn \cdot rm) + ra$. The `smmls` rd, rn, rm, ra instruction computes $rd \leftarrow \left\lfloor \frac{ra \cdot 2^{32} - rn \cdot rm}{2^{32}} \right\rfloor$ and we will discuss it again in Section 5.3.

5.3 Optimized polynomial multiplication

This section presents optimized polynomial multiplication on Cortex-M4.

5.3.1 Polynomial multiplication for negative polynomials

In-place double modular reductions. We highlight the critical role of in-place double modular reductions in Raccoon and underscore the effectiveness of Seiler's Montgomery reduction for achieving such an implementation. The function poly_split() in Raccoon requires two consecutive 32-bit modular reductions modulo q_1 and q_2 for each 64-bit polynomial coefficient $a = a_0 + a_1 \cdot 2^{32}$. In this context, the in-place double modular reduction involves computing two reductions, $a'_0 = a \bmod q_1, a'_1 = a \bmod q_2$, and storing the results a'_0 and a'_1 directly in the memory locations originally holding a_0 and a_1. The state-of-the-art approach for modular reduction of double-word integers in LBC relies on Montgomery reduction [86], which will be the primary focus of this discussion.

Original Montgomery Reduction. The state-of-the-art 32-bit Montgomery reduction on Cortex-M4, as detailed in Algorithm 32, was presented in [57]. However, this implementation modifies the input values a_0 and a_1 after Step 2 in Algo-

Algorithm 32 The state-of-the-art 32-bit Montgomery reduction [57]

Input: The 64-bit coefficient $a = a_0 + a_1 \cdot 2^{32}$

Output: $a \cdot 2^{-32} \bmod^{\pm} q_i$

1: `mul` $t, a_0, -q_i'$

2: `smlal` a_0, a_1, t, q_i

3: **return** a_1

rithm 32, preventing the second reduction from being performed in-place efficiently. Consequently, to execute the second Montgomery reduction for $a = a_0 + a_1 \cdot 2^{32}$, it is necessary to pre-save the input a_0 and a_1 into two additional registers (as seen in Steps 2 and 3 in Algorithm 33) before performing the Montgomery reduction modulo q_1. Moreover, to achieve an in-place implementation, an additional `mov` instruction is required to transfer the result back to a_0. As shown in Algorithm 33, performing in-place double modular reductions with original Montgomery reduction requires seven instructions and three intermediate registers, making it both computationally slow and register-intensive.

Seiler's Montgomery Reduction. This limitation motivates the adoption of Seiler's variant of Montgomery reduction (Seiler's Montgomery reduction), as proposed in [104], for implementing double modular reductions. The Seiler's variant, illustrated in Algorithm 34, offers a significant advantage: the final subtraction operation in Montgomery reduction depends solely on a_1. This eliminates the need to rely on a_0 after Step 1 and 2 in Algorithm 34. In this context, during Steps 3 through 6, only the most significant halves of $t_1 \times q_1$ and $t_2 \times q_2$ need to be processed, allowing subtraction from a_1 independently. This sequence can be executed entirely in-place. As a result, Seiler's Montgomery reduction enables double modular reductions using only six instructions and two intermediate registers—one fewer instruction and one fewer register compared to the original Montgomery reduction.

`SMMLS vs SMMUL & SUB.` During the optimization process of this algorithm, we identified a subtle difference between the ARM Cortex-M4 instructions `smmls` and the combination of `smmul` and `sub`. This distinction is documented here to prevent

future researchers from encountering similar challenge.

According to the "Cortex-M4 Devices Generic User Guide"[1], the description of `smmls` aligns with performing `smmul` followed by `sub`. For example, Steps 3∼6 of Algorithm 34 should theoretically be further optimized by replacing these steps with `smmls` a_0, t_1, q_1, a_1 and `smmls` a_1, t_2, q_2, a_1. However, we observed that the results obtained using `smmls` were smaller by 1 compared to those produced by executing `smmul` and `sub` separately, when the low half of the product $t_1 \times q_1$ is non-zero. Upon further investigation, we discovered on the ARM developer website[2] that the `smmls` a_0, t_1, q_1, a_1 instruction computes:

$$a_0 = \left\lfloor \frac{a_1 \cdot 2^{32} - t_1 \times q_1}{2^{32}} \right\rfloor, \tag{5.3.1}$$

rather than

$$a_0 = a_1 - \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor, \tag{5.3.2}$$

which is the desired result for Seiler's Montgomery reduction and is achieved by separately using `smmul` followed by `sub`. The discrepancy arises because the low half of the product $t_1 \times q_1$ is generally non-zero while the lower 32 bits of $a_1 \cdot 2^{32}$ are all zero, causing the low half of the 64-bit subtraction $a_1 \cdot 2^{32} - t_1 \times q_1$ in Equation 5.3.1 to produce a borrow bit to the upper 32 bits. This explains why results from `smmls` are mostly smaller by 1 compared to those computed with `smmul` and `sub`. Consequently, `smmls` and the combination of `smmul` and `sub` are not interchangeable. The design of `smmls` renders it unsuitable for further optimizing Seiler's Montgomery reduction.

[1] `https://developer.arm.com/documentation/dui0553/latest/`

[2] `https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/multiply-instructions/smmul--smmla--and-smmls`

Algorithm 35 The proposed negative double Montgomery reductions

Input: The 64-bit coefficient $a = a_0 + a_1 \cdot 2^{32}$, moduli $q_1, q_2, q_1' = q_1^{-1} \bmod 2^{32}, q_2' = q_2^{-1} \bmod 2^{32}$

Output: $-a \cdot 2^{-32} \bmod^{\pm} q_1, -a \cdot 2^{-32} \bmod^{\pm} q_2$

1: `mul` t_1, a_0, q_1'
2: `mul` t_2, a_0, q_2'
3: `neg` a_1, a_1
4: `smmla` a_0, t_1, q_1, a_1
5: `smmla` a_1, t_2, q_2, a_1
6: **return** a_0, a_1

Algorithm 33 Double modular reductions with original Montgomery reduction

Input: $a = a_0 + a_1 \cdot 2^{32}$

Output: $a \cdot 2^{-32} \bmod^{\pm} q_1, a \cdot 2^{-32} \bmod^{\pm} q_2$

1: `mul` $t_3, a_0, -q_2'$
2: `mov` t_1, a_0
3: `mov` t_2, a_1
4: `smlal` a_0, a_1, t_3, q_2
5: `mul` $t_3, t_1, -q_1'$
6: `smlal` t_1, t_2, t_3, q_1
7: `mov` a_0, t_2
8: **return** a_0, a_1

Algorithm 34 Double modular reductions with Seiler's Montgomery reduction

Input: $a = a_0 + a_1 \cdot 2^{32}$

Output: $a \cdot 2^{-32} \bmod^{\pm} q_1, a \cdot 2^{-32} \bmod^{\pm} q_2$

1: `mul` t_1, a_0, q_1'
2: `mul` t_2, a_0, q_2'
3: `smmul` t_1, t_1, q_1
4: `smmul` t_2, t_2, q_2
5: `sub` a_0, a_1, t_1
6: `sub` a_1, a_1, t_2
7: **return** a_0, a_1

Proposed Negative Double Montgomery reductions. To address the issues with `smmls` and further optimize double Montgomery reductions, we propose a faster implementation, referred to as negative double Montgomery reductions. This approach computes the negatives of the desired results:

$$a_0' = -a \cdot 2^{-32} \bmod^{\pm} q_1, a_1' = -a \cdot 2^{-32} \bmod^{\pm} q_2. \tag{5.3.3}$$

The negative variant of modular reduction does not impact the subsequent NTT

or Raccoon computations, as will be demonstrated later. Racall that in the `smmls`, the 64-bit subtraction $a_1 \cdot 2^{32} - t_1 \times q_1$ introduces a borrow bit that compromises the correctness of the operation. This occurs because the low 32 bits of $a_1 \cdot 2^{32}$ are all zeros, making the subtraction sensitive to the borrow. To circumvent this issue, we reverse the operation by subtracting $a_1 \cdot 2^{32}$ from $t_1 \times q_1$, avoiding the errors caused by the borrow bit. In our proposed method, we first negate a_1 (Step 3 of Algorithm 35), followed by using the `smmla` instruction: $a_0 = \texttt{smmla}(t_1, q_1, -a_1)$. This instruction effectively computes the addition of the negated a_1 and yields:

$$a_0 = \left\lfloor \frac{t_1 \times q_1 + (-a_1 \cdot 2^{32})}{2^{32}} \right\rfloor$$

Expanding this result:

$$a_0 = \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor + (-a_1) = -\left(a_1 - \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor \right). \tag{5.3.4}$$

This demonstrates that the sequence is equivalent to the negative of the desired result of Seiler's Montgomery reduction (Equation 5.3.2). Compared to the two variants in Algorithm 33 and Algorithm 34, Algorithm 35 is 2 instructions and 1 instruction faster than the two implementations in Algorithm 33 and Algorithm 34, respectively.

NTT for negative polynomial. The poly_split() function produces a negative polynomial. To handle the polynomial multiplication for negative polynomials using NTT, we recall the fundamental linearity property of Discrete Fourier transform (DFT) and NTT:

Property 1 (Linearity of DFT [54])**.** *Let a and b be complex numbers, and let x and y be n-point signals such that* $\mathrm{DFT}(x) = \hat{x}$ *and* $\mathrm{DFT}(y) = \hat{y}$*. Then, the DFT satisfies the linearity property:* $\mathrm{DFT}(ax + by) = a\hat{x} + b\hat{y} = a\hat{x} + b\hat{y}$.

Property 2 (Linearity of NTT [5])**.** *Let $a, b \in \mathbb{Z}_q$, and let x and y be polynomials in the polynomial ring R_q such that* $\mathrm{NTT}(x) = \hat{x}$ *and* $\mathrm{NTT}(y) = \hat{y}$*. Then, the NTT satisfies:* $\mathrm{NTT}(ax + by) = a\hat{x} + b\hat{y}$.

Proof of Property 2. The correctness of both Property 1 and Property 2 follows from the linearity of the transform. To demonstrate, we use the example of polynomial multiplication in $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ as adopted in Raccoon. Let ζ be the $2n$-th root of unity in $\mathbb{Z}_q$, and let $\omega = \zeta^2$ be the corresponding n-th root of unity in $\mathbb{Z}_q$. The forward NTT computation for the polynomial $ax + by$ can be written as:

$$\begin{aligned}
\mathrm{NTT}(ax + by)_j &= \sum_{i=0}^{n-1} (ax_i + by_i)\zeta^i \omega^{ij} \bmod q \\
&= \sum_{i=0}^{n-1} (ax_i + by_i)\zeta^{2ij+i} \bmod q \\
&= \sum_{i=0}^{n-1} (ax_i)\zeta^{2ij+i} \bmod q + \sum_{i=0}^{n-1} (by_i)\zeta^{2ij+i} \bmod q \\
&= a\sum_{i=0}^{n-1} x_i\zeta^{2ij+i} \bmod q + b\sum_{i=0}^{n-1} y_i\zeta^{2ij+i} \bmod q \\
&= a\hat{x}_j + b\hat{y}_j \bmod q, j = 0, 1, \ldots, n-1.
\end{aligned} \tag{5.3.5}$$

This completes the proof. □

Now, consider the case of a negative polynomial $-x$. The NTT computation for $-x$ is simply a special case of Property 2 with $a = -1$ and $b = 0$. Therefore, we obtain the following relation: $\mathrm{NTT}(-x) = -\hat{x} = -\mathrm{NTT}(x)$. This property holds due to the linearity of the NTT, meaning that negating the polynomial simply negates the result of the NTT computation.

Recall that the polynomial multiplication of a and b in R_q using NTT is computed as $c = \mathrm{INTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$, where $\circ$ denotes pointwise multiplication of the NTT-transformed elements. For two negative polynomials $-a$ and $-b$, the polynomial multiplication using NTT is performed as:

$$\begin{aligned}
c &= \mathrm{INTT}(\mathrm{NTT}(-a) \circ \mathrm{NTT}(-b)) \\
&= \mathrm{INTT}((-\mathrm{NTT}(a)) \circ (-\mathrm{NTT}(b))) \\
&= \mathrm{INTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b)).
\end{aligned}$$

In this case, the pointwise multiplication of $\mathrm{NTT}(-a)$ and $\mathrm{NTT}(-b)$ naturally cancels out the negative signs, and the subsequent INTT computation is performed as

usual.

Other adaptations for negative polynomials During Raccoon's key generation and signature generation, we also need to handle the polynomial multiplication of a negative polynomial and a normal polynomial. Consequently, it is essential to keep track of negative polynomials in the signature generation and utilize the associated components in Raccoon to properly manage the negative signs.

For example, during the computation of $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$ in Step 16 of Algorithm 12, the use of the proposed double modular reductions (Algorithm 35) and NTT for negative polynomials results in the NTT representations $-\hat{c}_{\mathsf{poly}}$, $-[\![\hat{\mathbf{s}}]\!]$ and $-[\![\hat{\mathbf{r}}]\!]$, where $[\![x]\!]$ denotes the mask form of the sensitive data x, as described in Subsection 2.1.1. While the pointwise multiplication of $(-\hat{c}_{\mathsf{poly}}) \circ (-[\![\hat{\mathbf{s}}]\!])$ cancels out the negative signs, it still requires subtracting $-[\![\hat{\mathbf{r}}]\!]$ to obtain the correct results. Instead of directly subtracting $-[\![\hat{\mathbf{r}}]\!]$, we observe that the Refresh() component is required to refresh $-[\![\mathbf{s}]\!]$ and $-[\![\mathbf{r}]\!]$ in Steps 12 and 13 of Algorithm 12. Because this component mainly consists of addition of $[\![\hat{\mathbf{0}}]\!]$, we find it perfect to handle the negative signs properly. Therefore, we propose two variants of the Refresh() component in Raccoon for the computation of $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$.

The first variant is computed as $\mathsf{Refresh}_1(-[\![\hat{\mathbf{r}}]\!]) = (-[\![\hat{\mathbf{0}}]\!]) + (-[\![\hat{\mathbf{r}}]\!]) = -([\![\hat{\mathbf{0}}]\!] + [\![\hat{\mathbf{r}}]\!])$, where the NTT-domain element $-[\![\hat{\mathbf{0}}]\!]$ is also optimized with the proposed double modular reductions described in Algorithm 35. This approach maintains the negative signs for the follow-up computations. The second variant is computed as $\mathsf{Refresh}_2(-[\![\hat{\mathbf{s}}]\!]) = [\![\hat{\mathbf{0}}]\!] - (-[\![\hat{\mathbf{s}}]\!]) = [\![\hat{\mathbf{0}}]\!] + [\![\hat{\mathbf{s}}]\!]$, where the NTT-domain element $[\![\hat{\mathbf{0}}]\!]$ is implemented using the double modular reductions with Seiler's Montgomery reduction, as shown in Algorithm 34. This variant converts the negative elements into positive elements. Using the above methods, the computation proceeds as: $-[\![\hat{\mathbf{z}}]\!] := -\hat{c}_{\mathsf{poly}} \cdot [\![\hat{\mathbf{s}}]\!] + (-[\![\hat{\mathbf{r}}]\!])$, and then in Step 15 of Algorithm 12, $\mathsf{Refresh}_2(-[\![\hat{\mathbf{z}}]\!])$ is used to eliminate the negative signs. It should be noted that these adaptations do not add extra cost to Raccoon's signature generation. On the contrary, the integration of the proposed double modular reductions with Seiler's Montgomery reduction

(Algorithm 35 and Algorithm 34) contributes to a further speedup of the Refresh() function.

NTT/INTT implementations. The NTTs/INTTs over q_1 and q_2 are computed following the in-place double modular reductions. Due to the register constraints of the Cortex-M4, which has only 14 general-purpose registers, performing the NTT/INTT computations for both q_1 and q_2 simultaneously is challenging, which would prevent us from implementing a 3-layer merging strategy, restricting us to a 2-layer merging strategy instead, thus increasing the number of memory accesses. As a result, we perform the two consecutive NTT/INTT computations sequentially, ensuring that all available registers are fully utilized to accelerate each NTT/INTT computation over q_i.

We adopt the 3-layer merging approach as in Dilithium [3] except that the 9-layer NTT computation in Raccoon is well-suited for the 3-layer merging strategy because a 3+3+3 layer merging strategy can be deployed for the computation without necessitating more memory access operations compared to the 8-layer NTT in Dilithium. The 32-bit modular multiplication by the twiddle factor in the butterfly units is implemented using Algorithm 32 as presented in [57]. Similar to [2, 3], we adopt the CT butterfly for both NTT and INTT. In contrast to the GS butterfly, using the CT butterfly helps limit the coefficient growth and minimize the number of modular reductions in INTT, which will be detailed below.

Lazy reduction in NTT/INTT. For Raccoon's $q = q_1 q_2 = 16515073 \times 33292289 = 549824583172097$, a 64-bit signed integer can accommodate absolute values up to $16775q$ while a 32-bit signed integer can accommodate values up to $130q_1$ and $64q_2$ for q_1 and q_2, respectively. To leverage the lazy reduction technique in NTT/INTT, we need to ensure that all of the polynomial coefficients will not exceed these bounds to ensure the correctness of the implementation.

As described before, the double modular reductions are required to split the 64-bit polynomial over $\mathbb{Z}_q$ to two 32-bit polynomials over $\mathbb{Z}_{q_i}$. This step ensures that all coefficients of the input polynomial are reduced to the range $(-q, q)$. As a result, it

is sufficient to verify that the input polynomial for the NTT stays within the bounds of Montgomery reduction, namely, $16775q$ for a 64-bit signed integer. Subsequently, the 9-layer 32-bit NTT over q_i is executed using the Cooley-Tukey (CT) butterfly structure. After the 9-layer NTT, the coefficients increase by at most $9q_i$. These coefficients can then be directly used in the subsequent matrix-vector multiplication, without the need for additional modular reductions during the NTT computation.

The CT butterfly algorithm is also employed in the INTT, as in [2]. The use of the CT algorithm in the INTT omits certain modular reductions, referred to as "light butterflies" [2]. The first three layers of INTT with light butterflies would expand $N/8$ of the coefficients by a factor of 8 and $N/8$ of the coefficients by a factor of 4. The subsequent three layers of light butterflies apply only to 64 coefficients and further expand 8 coefficients by a factor of 8 and 8 coefficients by a factor of 4.

The input polynomial for the INTT typically originates from the matrix-vector multiplication, which produces polynomials with maximum coefficients in absolute value of up to $l \cdot q_i$, namely $4q_i$, $5q_i$, and $7q_i$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. After the first three layers of the INTT, the coefficients may increase to $32q_i$, $40q_i$, and $56q_i$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. These values remain below the maximum bounds, $130q_1$ and $64q_2$. However, to apply the second three layers of light butterflies for the 64 coefficients, these coefficients must first be reduced using Barrett reduction. The remaining INTT computations, utilizing the standard CT butterfly algorithm, can then be executed safely without exceeding the maximum bounds. Overall, the INTT computation with CT algorithm only requires modular reductions for 64 coefficients for each q_i.

After INTT, the CRT computation is required to combine two 32-bit polynomials over $\mathbb{Z}_{q_i}$ to the 64-bit polynomial over $\mathbb{Z}_q$. Let c denote the 64-bit coefficient in the 64-bit polynomial over $\mathbb{Z}_q$, and $c_1 = c \bmod q_1$ and $c_2 = c \bmod q_2$ be two coefficients that need to be combined in the CRT computation. The CRT computation proceeds as follows: $c = c_1 q_2 (q_2^{-1} \bmod q_1) + c_2 q_1 (q_1^{-1} \bmod q_2) \bmod q$. Similar

to the reference implementation of Raccoon [95], the CRT computation is carried out together with the multiplication of $n^{-1} \bmod q_i$ at the end of two 32-bit INTT computations. Specifically, we can precompute $\zeta_1 = \zeta \cdot (q_2 n)^{-1} \cdot (2^{32})^4 \bmod q_1$ and $\zeta_2 = \zeta \cdot (q_1 n)^{-1} \cdot (2^{32})^4 \bmod q_2$ to merge the multiplication of q_i^{-1}, n^{-1} and the normalization factor $(2^{32})^4$ into a single Montgomery multiplication at the final layer of the INTT, i.e., $c'_i = c_i \cdot \zeta_i$. The final modular multiplications in the INTT will reduce all coefficients back to the range $(-q_i, q_i)$. The subsequent CRT computation, $c = c'_1 q_2 + c'_2 q_1$, can be efficiently implemented using the `smull` and `smmla` instructions, producing coefficients in the range $(-2q, 2q)$.

5.3.2 Lazy reduction for masking components

In this section, we demonstrate that lazy reduction can also be applied to the masking components of Raccoon, including `ZeroEncoding`, `Refresh`, `Decode`, and `AddRepNoise`. These components in Raccoon's reference implementation involve numerous conditional additions/subtractions to perform modular subtractions/additions, ensuring that the results remain in the range $[0, q)$. We provide a detailed range analysis to demonstrate that the lazy reduction technique can be applied throughout the Raccoon implementation, eliminating many of these conditional operations. By postponing modular reductions until absolutely necessary, we significantly optimize Raccoon's performance, reducing unnecessary computational overhead while maintaining the correctness of the implementation.

Time complexity and output range analysis. We now conduct a thorough time complexity and output range analysis to highlight the improvements gained by using lazy reduction in the masking components of Raccoon while ensuring that all computations are correctly handled without overflow. The time complexity here mainly indicates the number of conditional additions/subtractions. We denote $[\![x]\!]$ as the input masked polynomial in R_q for output range discussion, and $[\![x_i]\!]$ as the input masked polynomial in R_{q_i}, where each $[\![x_i]\!]$=$[\![x]\!] \bmod q_i$ for $i = 1, 2$.

ZeroEncoding. The `ZeroEncoding` subroutine encodes zero into d-sharing and generates the masked polynomial $[\![0]\!]$. This is achieved by recursively sampling $d/2$- and $d/2$-sharing for the left and right sides, then adding and subtracting newly sampled polynomials in $\mathbb{Z}_q$ to ensure that the sum of the d-sharing $[\![0]\!]$ is equal to zero. In the reference implementation of Raccoon, expensive conditional additions and subtractions by q are used for n coefficients to reduce each coefficient modulo q during polynomial addition and subtraction across the system. This significantly increases the time complexity.

By leveraging lazy reduction in `ZeroEncoding`, we reduce the time complexity. Specifically, the lazy reduction eliminates $2nd \cdot \log(d)$ conditional additions/subtractions by q when $d > 2$. For $d = 2$, n conditional additions by q are eliminated entirely. When $d = 1$, the encoding of zero is straightforward, and no further conditional additions or subtractions are necessary, leading to no optimization.

Using the lazy reduction technique, the output of this subroutine is bounded by $q \cdot \log(d)$. For the maximum value of $d = 32$ in Raccoon, the `ZeroEncoding` subroutine produces results with an absolute value of at most $5q$. This value comfortably fits within a 64-bit signed integer and does not affect the correctness of subsequent computations.

Refresh. The `Refresh` subroutine utilizes the `ZeroEncoding` component to generate a masking $[\![0]\!]$ and computes a fresh d-sharing polynomial $[\![x]\!]' = [\![x]\!] + [\![0]\!]$. There are two variants of the `Refresh` subroutine depending on whether the polynomial $[\![x]\!]$ is in R_q or $[\![x_i]\!]$ is in R_{q_i}.

The lazy reduction technique significantly reduces the time complexity of the `Refresh` subroutine. Specifically, the technique eliminates nd conditional subtractions by q, or $2nd$ conditional subtractions by q_i, excluding the `ZeroEncoding` component. When including the `ZeroEncoding` subroutine, the total reduction is $nd + 2nd \cdot \log(d)$ conditional additions/subtractions by q, or $2nd + 4nd \cdot \log(d)$ conditional additions/subtractions by q_i in total. This substantial reduction in modular operations directly translates into improved performance.

Using the lazy reduction technique for these two variants produces results within different output ranges:

1. **When $[\![x]\!]$ is in R_q**: The polynomial addition $[\![x]\!]' = [\![x]\!] + [\![0]\!]$ is performed directly, resulting in coefficients with an absolute value of at most $|x|+q\cdot\log(d)$.

2. **When $[\![x_i]\!]$ is in R_{q_i}**: The double modular reductions described in Subsection 5.3.1 are first applied to reduce all coefficients of $[\![0]\!]$ modulo q_1 and q_2. This step ensures that all coefficients of $[\![0]\!]$ are reduced to the range $(-q, q)$ before the polynomial addition. In this case, the output range is limited to an absolute value of at most $|x_i| + q_i$.

AddRepNoise. The `AddRepNoise` subroutine implements the Sum of Uniform (SU) distribution in a masked fashion, which involves repeatedly sampling noise of length $|u_t|$ or $|u_w|$ and adding this noise to the masked polynomial $[\![x]\!]$. The function then refreshes $[\![x]\!]$ using the `Refresh` subroutine, iterating rep times. This addition can also be carried out using the lazy reduction technique, which reduces $nd \cdot \text{rep}$ conditional subtractions by q.

The output range of the `AddRepNoise` function is determined by the sum of the initial value of $[\![x]\!]$ and the noise added across rep iterations. More precisely, the output range for $[\![\mathbf{s}]\!]$ and $[\![\mathbf{t}]\!]$ in the keygen algorithm (from `keygen`) is given by: $|x| + \text{rep}\cdot(2^{u_t} + q \cdot \log(d))$, and for $[\![\mathbf{r}]\!]$ and $[\![\mathbf{w}]\!]$ in the sign algorithm (from `sign`), it is given by: $|x| + \text{rep}\cdot(2^{u_w} + q \cdot \log(d))$. The maximum value of these expressions occurs when $d = 32$, leading to the following upper bound for the output range: $|x| + 4 \cdot (2^{41} + 5q) < |x| + 21q$. This result is well within the safe range of a 64-bit signed integer, as it is considerably smaller than the upper bound $16775q$. Thus, the `AddRepNoise` function can be executed safely with minimal risk of overflow

Decode. The `Decode` subroutine in Raccoon is responsible for decoding a masked polynomial $[\![x]\!]$ by adding all shares together and recovering the unmasked value x in R_q. Similar to the `Refresh` subroutine, there are two variants of the `Decode` subroutine, depending on whether the polynomial $[\![x]\!]$ is in R_q or R_{q_i}. By applying

Table 5.1: Raccoon parameters [95]

	Raccoon-128						Raccoon-192						Raccoon-256					
d	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
k	5	=	=	=	=	=	7	=	=	=	=	=	9	=	=	=	=	=
l	4	=	=	=	=	=	5	=	=	=	=	=	7	=	=	=	=	=
rep	8	4	2	4	2	4	8	4	2	4	2	4	8	4	2	4	2	4
u_t	6	6	6	5	5	4	7	7	7	6	6	5	6	6	6	5	5	4
u_w	41	41	41	40	40	39	41	41	41	40	40	39	41	41	41	40	40	39

k: Number of rows in $\mathbf{A}$, length of vector $\mathbf{t}$; l: Number of columns in $\mathbf{A}$, length of vector $\mathbf{s}$; d: Number of masking shares; rep: Number of iterations for SU distribution; u_t: Distribution parameter for $\mathbf{s}$ and $\mathbf{t}$; u_w Distribution parameter for $\mathbf{r}$ and $\mathbf{w}$.

the lazy reduction technique, the number of conditional subtractions by q is reduced by $n \cdot (d-1)$ for the case where $[\![x]\!]$ is in R_q, or by $2n \cdot (d-1)$ when $[\![x]\!]$ is in R_{q_i}.

The output range of the `Decode` subroutine is determined by the number of shares, and it is given by:

1. For $[\![x]\!]$ in R_q, the output range is at most $d \cdot |x|$.

2. For $[\![x_i]\!]$ in R_{q_i}, the output range is at most $d \cdot |x_i|$.

For the polynomial $[\![x_i]\!]$ in R_{q_i}, the coefficients are stored in 32-bit signed integers. Depending on the values of the coefficients, there is a risk of overflow, especially for high-order Raccoon computations where the coefficients may grow larger during subsequent operations. This potential overflow needs careful attention to avoid errors or incorrect results after the `Decode` subroutine.

Table 5.2 summarizes the complexity reduction and output range for four Raccoon subroutines. The first and second rows of the `Refresh` and `Decode` subroutines correspond to the cases where the masked polynomial $[\![x]\!]$ is in R_q or $[\![x_i]\!]$ is in R_{q_i}, respectively. In summary, the lazy reduction strategy significantly reduces time complexity, thereby enhancing performance. Furthermore, as will be demonstrated in Section 5.3.2, the output range produced using this approach comfortably fits within the 32-bit or 64-bit maximum bounds, without introducing many side effects

Table 5.2: Complexity reduction and output range using the lazy reduction

	# of conditional operation	Output range (absolute value)
`ZeroEncoding`	$2nd \cdot \log(d)$	$q \cdot \log(d)$
`Refresh`	$nd + 2nd \cdot \log(d)$	$\|x\| + q \cdot \log(d)$
	$2nd + 4nd \cdot \log(d)$	$\|x_i\| + q_i$
`AddRepNoise`	$nd \cdot \text{rep}$	$\|x\| + \text{rep} \cdot (2^{u_w} + q \cdot \log(d))$
`Decode`	$n \cdot (d-1)$	$d \cdot \|x\|$
	$2n \cdot (d-1)$	$d \cdot \|x_i\|$

in the overall Raccoon's implementation.

Range analysis in `keygen` and `sign`. We now conduct a thorough range analysis to ensure that, when using lazy reduction, most of the Raccoon computations are performed correctly without causing overflow. Only minimal additional modular reductions are required to guarantee the correctness of the implementation.

For the keygen algorithm, the `ZeroEncoding` and `AddRepNoise` subroutines in Steps 3 and 4 of Algorithm 10 generate the masked polynomial $[\![\mathbf{s}]\!]$ within the range $q \cdot \log(q) + \text{rep} \cdot (2^{u_t} + q \cdot \log(d))$, which is smaller than $26q$ and comfortably fits within a 64-bit signed integer. As described in the lazy reduction for NTT/INTT in Section 5.3.1, double modular reductions can first reduce $[\![\mathbf{s}]\!]$ to the range $(-q_i, q_i)$, allowing the NTT computation to proceed without further modular reductions. The matrix-vector multiplication in Step 5 of Algorithm 10 produces $[\![\mathbf{t}]\!]$ within the range $(-lq_i, lq_i)$, which can also be directly input into INTT, as detailed in the lazy reduction of INTT in Section 5.3.1. The INTT then outputs results in the range $(-2q, 2q)$. The `AddRepNoise` and `Decode` functions will further expand the coefficients of $\mathbf{t}$ to a range of $d \cdot (2q + \text{rep} \cdot (2^{u_t} + q \cdot \log(q)))$. For the maximum values of $d = 32$, $\text{rep} = 4$, and $u_t = 5$ as shown in Table 5.1, this results in coefficients smaller than $736q$, which is well below the maximum bound $16775q$ for a 64-bit signed integer. After the `Decode` subroutine, only one Barrett reduction is required

to reduce $\mathbf{t}$ modulo q for subsequent computations. Using lazy reduction in `keygen` allows us to fully exploit the redundancy in 32-bit or 64-bit signed integers, thereby minimizing the number of modular reductions required.

For the sign algorithm, the computations in `sign` are the most complex in Raccoon and require a more careful range analysis to ensure correctness. Steps 4 through 9 of Algorithm 12 are similar to those in Algorithm 10, with the main difference being that the parameter for `AddRepNoise` is u_w instead of u_t in `keygen`. This does not impact the range analysis, and the maximum bound of the produced $\mathbf{w}$ remains smaller than $736q$. Therefore, the range analysis for Steps 4 through 9 is identical to that in `keygen`.

The critical operations occur in Steps 12 through 18. The `Refresh` functions in Steps 12 and 13 are executed in the NTT domain, where $[\![\mathbf{s}]\!]$ and $[\![\mathbf{r}]\!]$ are split into elements over $\mathbb{Z}_{q_i}$, already being in the NTT domain. As a result, the maximum value of the coefficients in $[\![\mathbf{s}]\!]$ and $[\![\mathbf{r}]\!]$ is less than $9q_i$. As shown in Table 5.2, this variant of the `Refresh` subroutine over $\mathbb{Z}_{q_i}$ generates an output range of $|x_i| + q_i$, i.e., $10q_i$ in this case. Coefficients in this range remain within the maximum bounds $130q_1$ and $64q_2$, ensuring they can be safely handled in the subsequent computations in Step 14 of Algorithm 12.

In Step 14, the polynomial multiplication and accumulation $c_{poly} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{t}]\!]$ first multiply $[\![\mathbf{r}]\!]$ by $2^{-32} \bmod q_i$ to maintain the same Montgomery domain as $c_{poly} \cdot [\![\mathbf{s}]\!]$. This reduces the maximum bound of $[\![\mathbf{r}]\!]$ to q_i. Then, Step 14 generates $[\![\mathbf{z}]\!]$ with coefficients in the range $(-2q_i, 2q_i)$. The `Refresh` function in Step 15 expands the maximum bound to an absolute value of $3q_i$. For $d \leq 16$, the `Decode` function in Step 16 produces results smaller than $48q_i$, which fit within 32-bit signed integers and require no modular reduction. For $d = 32$, however, the `Decode` function generates results within $96q_i$. When using modulus q_2, this could exceed the maximum bound $64q_2$, necessitating an additional Barrett reduction in the `Decode` function to prevent overflow. After the `Decode` function, a final Barrett reduction is required to reduce $\mathbf{z}$, enabling it to be passed to INTT and then serialized as the signature.

Algorithm 36 Left-shift operation by 42 in Raccoon

Input: The 64-bit coefficient $a = a_0 + a_1 \cdot 2^{32}$

Output: $a << 42$

1: `lsls` $a_1, a_0, \#10$

2: `movs` $a_0, \#0$

3: **return** $a = a_0 + a_1 \cdot 2^{32}$

5.3.3 Tailored implementations for Raccoon

Apart from the optimizations discussed earlier, we also introduce additional improvements specifically tailored for Raccoon's parameters.

Polynomial shifting. Some operations in Raccoon are performed over the polynomial ring R_q with a 49-bit modulus, which may involve 64-bit polynomial operations. On 32-bit platforms, certain 64-bit operations, such as right/left shifting, can be expensive due to the need to carefully manage results for both the high and low halves of the 64-bit value. However, we observe that the shifting lengths in Raccoon are limited to $v_t = 42$ and $v_w = 44$. This observation allows us to optimize these operations specifically for v_t and v_w. As shown in Algorithm 36, the left-shift operation by v_t can be accomplished using just two instructions, which offers a significant performance improvement over the reference implementation that uses variable shifting lengths. For comparison, the ARM compiler[3] with -O3 flag typically translates the left-shift of a 64-bit value into 8 instructions. Similar speed ups can be obtained for right-shift operations by 42 and 44 in Raccoon.

Coefficient reduction. Due to the throughout usage of lazy reduction in Raccoon, the coefficient reduction is still required to reduce the coefficients of polynomials in R_q or in R_{q_i} down to a small range so that the subsequent computations will not overflow. We adopt the modular reduction presented in the reference implementation of Dilithium and tailor it for the moduli q and q_i of Raccoon. For 32-bit modular reduction modulo q_i, the instruction sequence is similar to the modular reduction for Dilithium's modulus, which only takes three cycles on Cortex-M4. The

[3] `https://godbolt.org/`

Algorithm 37 The 64-bit modular reduction modulo q of Raccoon

Input: The 64-bit coefficient $a = a_0 + a_1 \cdot 2^{32}$ with $|a| \leq 2^{63} - 2^{48} - 1$ and 64-bit modulus $q = q_0 + q_1 \cdot 2^{32}$.

Output: $r = a \mod q \in (-q, q)$

1: `add` $t_0, a_1, \#2^{16}$

2: `asr` $t_0, t_0, \#17$

3: `smull` t_1, t_2, t_0, q_0

4: `mla` t_2, t_0, q_1, t_2

5: `subs` a_0, t_1

6: `sbc` a_1, t_2

7: **return** $r = a_0 + a_1 \cdot 2^{32}$

instruction sequence of the 64-bit modular reduction modulo q is shown in Algorithm 37. It only needs six instructions and outputs results in $(-q, q)$. A 64-bit conditional addition by q is required to reduce it to $[0, q)$ for the following computations in Raccoon. On the contrary, the 32-bit modular reduction modulo q_i does not necessarily need to reduce it to $[0, q_i)$ and can be achieved in a lazy form.

5.4 Enabling high-order Raccoon on IoT devices

Raccoon provides high-order masked implementations with up to 32 shares. These implementations require splitting each secret polynomial into d-share polynomials, which can lead to significant memory consumption. As a result, high-order masking implementations are not feasible on memory-constrained devices. For instance, the reference implementations of Raccoon with $d \geq 16$, Raccoon-192 with $d \geq 8$ and Raccoon-256 with $d \geq 4$ cannot run on the selected platform with only 640 KiB of RAM. Therefore, memory optimizations for high-order Raccoon are essential to ensure both protection against side-channel attacks and the practicality of Raccoon on such devices.

In this section, we propose several memory optimizations aimed at reducing

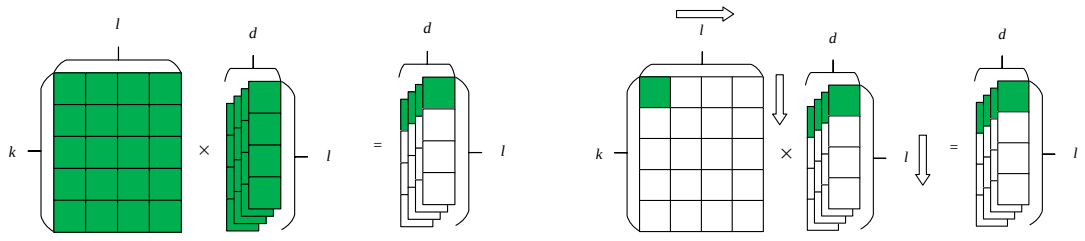
the memory consumption of Raccoon, thus enabling high-order Raccoon to provide stronger side-channel protection on memory-constrained devices. The memory consumption of the `sign` algorithm is the major obstacle hindering its practical deployment on such platforms. Therefore, we mainly focus on reducing the memory usage of `sign` in this section. The memory optimizations for `keygen` and `verify` can be carried out similarly.

5.4.1 Streaming the matrix-vector multiplication

The most memory-consuming operation in Raccoon is the matrix-vector multiplication with the d-share polynomial vector. As shown in Figure 5.1a, the green entries in the polynomial matrix and vector represent the actual memory used during the matrix-vector multiplication in Raccoon's reference implementation. Each n-degree polynomial in R_q consumes 4 KiB of stack memory. The polynomial matrix $\mathbf{A}$ therefore accounts for $4kl$ KiB; the secret polynomial vector $[\![\mathbf{r}]\!]$ consumes $4ld$ KiB; and the d-share polynomial $[\![\mathbf{w}]\!]$ accounts for $4d$ KiB. The polynomial matrix and the d-share polynomial vector are the main reason of the large stack usage. Both of them need to be used twice in Algorithm 12. Consequently, time-memory trade-offs are required to enable high-order Raccoon on memory-constrained devices.

Streaming the matrix A. Instead of retaining the entire matrix, one can generate the matrix $\mathbf{A}$ on-the-fly and store only a single entry at a time, following the approach proposed in [57]. This requires generating the matrix twice, in Steps 6 and 17 of Algorithm 12, but it can save 80 KiB, 140 KiB, and 252 KiB of memory for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. We apply this strategy to Raccoon-192 with $d = 8$ and Raccoon-256 with $d = 4$. These variants of Raccoon can be deployed successfully on the selected platform without significant performance degradation, as shown in Table 5.5.

Streaming the d-share vector $[\![\mathbf{r}]\!]$. As d increases, the stack usage of $[\![\mathbf{r}]\!]$ and $[\![\mathbf{w}]\!]$ grows linearly. When $d > k$, the stack usage of the d-share polynomial vector



(a) The reference implementation (b) The proposed implementation

Figure 5.1: The matrix-vector multiplication implementations of $\mathbf{A}\times[\![\mathbf{r}]\!]=[\![\mathbf{w}]\!]$ in the sign of Raccoon

$[\![\mathbf{r}]\!]$ surpasses that of $\mathbf{A}$. Consequently, streaming $[\![\mathbf{r}]\!]$ becomes essential to reduce memory consumption in Raccoon. As shown in Figure 5.1b, we generate only one d-share polynomial at a time, which reduces memory usage by $4(l-1)d$ KiB. It is important to note that the `AddRepNoise` subroutine requires generating κ secret key bits to sample a secret polynomial noise and add it to $[\![\mathbf{r}]\!]$ in Steps 5 and 7 of Algorithm 12. The reference implementation generates these bits using random bit generators within the `AddRepNoise` subroutine. However, this approach does not guarantee the same secret key bits, which is necessary to regenerate $[\![\mathbf{r}]\!]$. To address this, we pre-generate all required secret key bits and reuse them to regenerate $[\![\mathbf{r}]\!]$ in Step 14. This strategy is employed for Raccoon and Raccoon-192 when $d > 8$, and for Raccoon-256 when $d > 4$.

5.4.2 Memory reuse

In addition to the matrix-vector multiplication, the internal structure of the secret key, public key, and signature in Raccoon also contributes significantly to memory usage. The public key consists of a k-dimensional polynomial vector $\mathbf{t}$, which occupies $4k$ KiB of memory. The secret key comprises the public key and a d-share, l-dimensional secret polynomial vector $[\![\mathbf{s}]\!]$, consuming $4ld$ KiB of memory. The signature consists of a k-dimensional hint $\mathbf{h}$ and an l-dimensional vector $\mathbf{z}$, requiring a total of $4(k+l)$ KiB of memory. The d-share secret key must remain intact to preserve security against side-channel attacks, as stated in [95, Section 2.5]; thus, the memory space for the secret key cannot be further reduced.

We carefully analyze the operation sequences in Raccoon's `sign` procedure and identify opportunities to reuse memory for the polynomial vectors in the public key and signature. For instance, we reduce the memory required for the l-dimensional $\mathbf{z}$ by using part of the memory allocated for $[\![\mathbf{r}]\!]$, which is no longer needed after Step 14. This optimization is only possible when $d > l$. Similarly, we can reuse $[\![\mathbf{r}]\!]$ for computing $\mathbf{y}$, and $\mathbf{w}$ for computing $\mathbf{h}$. As a result, we reduce the memory for the polynomial vectors in the public key and signature by $(8k + 4l)$ KiB, making it feasible to deploy high-order Raccoon on memory-constrained devices.

5.5 Results and comparisons

The experimental setup is the same as the pqm4 [71]. The target Cortex-M4 board is the latest platform supported in pqm4, the NUCLEO_L4R5 board, with 2MB of flash and 640KiB of RAM. We use the optimized Keccak implementation presented in Section 4.6 for benchmarking the proposed implementations. The masking random number generator we used is the fast generator based on LFSR provided in the reference implementation.

5.5.1 Speed performance of polynomial arithmetic

Table 5.3 presents the results of the polynomial arithmetic for Raccoon-128 on the Cortex-M4. Compared to the reference implementation in [95], we achieve a comprehensive improvement in the performance of polynomial arithmetic. The polynomial splitting operation involves the double modular reductions discussed in Section 5.3.1. As shown, the proposed double modular reductions in Algorithm 34 and Algorithm 35 outperform the implementation based on the state-of-the-art Montgomery reduction in [57] by factors of 1.57× and 1.71×, respectively. The polynomial joining operation, which includes the CRT computation, outperforms the reference implementation by a factor of 1.63×. The performance improvements for NTT and INTT are even more significant, with speedups of 2.54× and 3.98×, respectively,

Table 5.3: Cycle counts (cc) of the polynomial arithmetic of Raccoon-128 on Cortex-M4.

	Split	Join	NTT	INTT	Left-shift	Right-shift	Add	Addq[a]
Ref. [95]	9795	22613	118455	171670	12371	14420	7236	10835
This work	6231 (5718)	13908	46677	43026	3801	4942	5970	9047
Ref/This work	1.57 × (1.71×)	1.63×	2.54×	3.98×	3.25×	2.93×	1.21 × (1.81×)	1.20×

[a] Addq denotes the polynomial addition with conditional subtraction of q.

Table 5.4: Cycle counts (cc) of the masking gadgets of Raccoon-128 on Cortex-M4.

		ZeroEncoding	AddRepNoise	Refresh	NTT Refresh	Decode	NTT Decode
$d = 1$	Ref. [95]	3643	4621694	55	53	7228	7228
	This work	3643	2838159	55	53	7227	7228
$d = 2$	Ref. [95]	19377	4785073	40878	68634	10836	14937
	This work	14005	2941116	25776	36666	5972	5714
$d = 4$	Ref. [95]	100749	4909375	144062	199455	32423	44725
	This work	71581	3028599	95438	117095	17827	17059
$d = 8$	Ref. [95]	326040	17286545	4621694	523182	75590	104296
	This work	230879	11084177	278300	321628	41534	39743
$d = 16$	Ref. [95]	900440	17783937	1073626	1294684	161901	223416
	This work	636435	11434030	731780	826092	88926	85086
$d = 32$	Ref. [95]	2297856	73123134	2644266	3086391	334528	461657
	This work	1622473	47134002	1813245	2001820	183711	178759

because we thoroughly leverage the layer merging and lazy reduction strategies to enhance efficiency. The 64-bit left-shift and right-shift operations are 3.25× and 2.93× faster, thanks to tailored implementations for Raccoon's parameters. The assembly implementation of polynomial addition and subtraction is also slightly faster than the reference implementation.

Additionally, we compared the performance of four subroutines optimized with the lazy reduction technique, as outlined in Subsection 5.3.2. The results, presented in Table 5.4, demonstrate that employing the lazy reduction strategy in these subroutines achieves speedups ranging from 1.38× to 1.81×. This optimization significantly enhances the overall performance of Raccoon.

We also compare four subroutines that we optimized with the lazy reduction

Table 5.5: Cycle counts (cc) and stack usage (Bytes) of keygen, sign, and verify of Raccoon on Cortex-M4. Averaged over 1000 iterations.

	Implementation	Raccoon-128 [a]			Raccoon-192			Raccoon-256		
		keygen	sign	verify	keygen	sign	verify	keygen	sign	verify
$d = 1$	Ref.[95]	29 073k	65 719k	21 851k	45 518k	94 450k	35 862k	73 878k	124 020k	60 837k
		83 232	230 752	111 960	107 864	290 815	144 800	140 704	505 320	185 832
	This work	19 637k	39 628k	13 226k	30 044k	56 658k	21 460k	47 631k	79 214k	36 098k
		82 584	230 104	111 248	107 232	332 568	144 152	140 040	504 664	185 184
$d = 2$	Ref.[95]	35 245k	72 595k	21 851k	53 705k	103 777k	35 858k	85 407k	136 329k	60 839k
		112 008	284 064	111 960	140 744	394 720	144 800	181 660	583 208	185 832
	This work	22 977k	43 196k	13 226k	34 448k	61 533k	21 458k	53 854k	85 741k	36 097k
		111 360	283 424	111 312	140 096	394 080	144 112	181 120	574 328	185 184
$d = 4$	Ref.[95]	46 043k	85 151k	21 849k	68 019k	108 992k	35 859k	-	-	-
		164 328	377 292	111 944	201 180	504 332	144 800	-	-	-
	This work	28 808k	50 052k	13 226k	42 113k	67 363k	21 460k	64 766k	216 313k	36 194k
		164 352	262 143	111 312	201 172	504 324	144 152	192 956	381 444	185 184
$d = 8$	Ref.[95]	111 445k	199 892k	21 852k	-	-	-	-	-	-
		262 636	299 007	111 960	-	-	-	-	-	-
	This work	76 364k	129 326k	13 226k	105 337k	295 197k	21 447k	150 611k	969 455k	36 193k
		262 636	557 604	111 312	266 848	488 072	144 152	344 696	504 648	185 192
$d = 16$	Ref.[95]	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-
	This work	100 786k	436 475k	13 284k						
		426 492	611 648	111 320						

[a] The first row of each entry indicates the cycle count and the second row refers to stack usage.

as described in Subsection 5.3.2. As shown in Table 5.4, using the lazy reduction strategy in these subroutines results in 1.38× to 2.61× speedups, which further improve the performance of Raccoon.

5.5.2 Performance of schemes

The cycle counts in Table 5.5 are measured by repeating each subroutines 1000 times, while the stack usage is measured by repeating 100 times, and take the averaged results. Combining all the aforementioned optimizations, including the efficient double modular reductions, lazy reduction for Raccoon subroutines, as well as the tailored optimizations for Raccoon operations, the proposed implementations reduce 32.46% $\sim$ 40.01% of the clock cycles compared to Raccoon's reference implementa-

tion. Moreover, the proposed memory optimizations enables the usage of high-order Raccoon, namely Raccoon-128 with $d = 16$, Raccoon-192 with $d = 8$, and Raccoon-256 with $d = 4, 8$ on the selected platform. Note that it could be extended to low-order Raccoon on platforms with more limited memory.

5.5.3 Comparison with masking Dilithium

Since Raccoon closely resembles Dilithium, Table 5.6 compares the performance of Raccoon-192 with the state-of-the-art high-order masking Dilithium3 [18] on the ARM Cortex-M4. Both Raccoon-192 and Dilithium3 provide the same security level. The results in [18] were obtained without accounting for the signature generation repetitions required by Dilithium. However, due to the rejection sampling procedure, Dilithium's signature generation typically requires approximately four repetitions. To ensure a fair comparison, we scale their results by a factor of 4 before comparing them with Raccoon-192.

As shown in the table, Raccoon-192 achieves speedups of 1.97×, 5.27×, and 3.51× over the masking Dilithium3 for masking orders $d = 2, 4, 8$, respectively. Although the unmasked Raccoon-192 implementation is significantly slower than the unmasked Dilithium3 implementation, we observe that as the masking order increases from 1 to 4, the performance of Dilithium3 deteriorates by a factor of 54.52×, whereas Raccoon-192 with $d = 4$ experiences only a 1.18× slowdown compared to Raccoon-192 with $d = 1$. This indicates that increasing the masking order in Raccoon incurs only linear complexity, making it more efficient in higher-order masking scenarios. The lower speedup at $d = 8$ is due to memory optimizations that slightly reduce Raccoon-192's performance. These results highlight Raccoon's superior efficiency in masked implementations compared to NIST's Dilithium, making it a promising LBC scheme for side-channel-resistant cryptographic applications.

Table 5.6: Comparison of high-order masking signature generation of Dilithium3 and Raccoon-192 on Cortex-M4.

Implementation	$d = 1$	$d = 2$	$d = 4$	$d = 8$
Raccoon-192	56 658k	61 533k	67 363k	295 197k
Dilithium3 [18]	6 523k[a]	121 016k	355 648k	1 038 972k
Dilithium3/Raccoon-192	0.12×	1.97×	5.27×	3.51×

a: Unmasked Dilithium3 implementation presented in Table 4.10.

5.6 Conclusion

This chapter presented the optimized implementations of the masking-friendly lattice-based cryptographic (LBC) scheme, Raccoon, on the Cortex-M4 platform. The proposed enhancements include efficient double modular reduction techniques, lazy reduction strategies for Raccoon subroutines, and tailored optimizations for Raccoon operations, resulting in significant improvements in its efficiency. Furthermore, memory optimization techniques for high-order Raccoon were explored, addressing the challenges posed by memory-constrained IoT devices. These optimizations enhance the feasibility and practicality of deploying high-order Raccoon, ensuring robust side-channel resistance in memory-limited devices.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

This thesis presented efficient, lightweight, and secure implementations of lattice-based cryptography (LBC) specifically tailored for Internet of Things (IoT) devices. The research addresses both the algorithmic design and implementation aspects to ensure practical deployment in resource-constrained environments. The main theoretical contribution of this thesis is the improvement of Plantard arithmetic introduced by Thomas Plantard in 2021, referred to as the improved Plantard arithmetic. Theoretical proofs of correctness for the improved Plantard arithmetic are provided from two perspectives, demonstrating its validity. We showed that the proposed modular arithmetic is particularly well-suited for NTT computations with word-size moduli and can replace the state-of-the-art Montgomery and Barrett arithmetic in the NTT computations of several LBC schemes, including the NIST PQC standards Kyber and Dilithium, on three 32-bit IoT platforms. The optimizations for Kyber and Dilithium presented in this thesis represent the current state-of-the-art implementations and have been integrated into the popular pqm4 repository. Additionally, this thesis proposed two optimization techniques—lazy rotations and memory access pipelining—to enhance the performance of Keccak, further improving the efficiency of PQC schemes that heavily rely on Keccak. The optimized implementations of Kyber, Dilithium, and Keccak on Cortex-M4 have been integrated

into the popular PQM4 repository [71], showcasing their practicality and promising efficiency. Moreover, side-channel secure cryptography, which protects implementations against side-channel attacks, is another important area explored in this thesis. The efficient, lightweight, and secure implementations of the masking-friendly LBC scheme, Raccoon, were studied. We also explored the memory optimizations for high-order Raccoon, addressing the challenges posed by memory-constrained IoT devices. These optimizations enhance the feasibility and practicality of deploying high-order Raccoon, ensuring robust side-channel resistance in memory-limited devices.

6.2 Future works

Plantard arithmetic on SIMD. The proposed improved Plantard arithmetic has been applied to several LBC schemes on 32-bit IoT platforms. However, its use on mainstream platforms with more powerful SIMD instruction sets, such as AVX2, AVX-512, and NEON, has been considered challenging due to the lack of efficient instructions for performing the $l \times 2l$-bit multiplication in Plantard arithmetic. Considering that Plantard arithmetic has a larger input range and smaller output range, it might be advantageous to limit coefficient growth in the NTT/INTT implementation, thus reducing the number of modular reductions. However, the performance gains from this approach may be limited and require further exploration. Additionally, there is potential for customizing vector instructions for Plantard arithmetic on RISC-V, which could help to further optimize the Plantard arithmetic for vector extension on RISC-V platforms.

LBC deployment on real-world applications. This thesis primarily focuses on optimizing the performance of LBC schemes on IoT devices. However, their deployment in real-world IoT applications remains an area for further exploration. Future work could investigate practical approaches for efficiently integrating and securing real-world IoT applications with quantum-safe LBC schemes and cryptographic protocols like IPSec, DNS, and TLS, addressing both performance and security concerns

in the evolving quantum era [27, 33].

Efficient implementation of NIST additional DSA candidates. NIST continues to call for the evaluation of additional digital signature algorithms to diversify the PQC DSA standards. The second round of the NIST additional DSA competition is underway, with 14 candidates across five categories advancing to the next stage. It remains promising to further explore their efficient implementation on various IoT devices. Additionally, it would be interesting to investigate whether the Plantard arithmetic can be applied to these schemes, such as the only LBC scheme, Hawk, in the second round.

BIBLIOGRAPHY

[1] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. Fast and clean: Auditable high-performance assembly via constraint solving. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):87–132, 2024.

[2] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):127–151, 2022.

[3] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022.

[4] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers new bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021.

[5] R. Agarwal and C. Burrus. Fast convolution using fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

[6] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The em side—channel (s). In *Cryptographic Hardware and Embedded Systems-*

CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4, pages 29–45. Springer, 2003.

[7] Google Quantum AI and Collaborators. Quantum error correction below the surface code threshold. *Nature*, 2024.

[8] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery.

[9] Gorjan Alagic, Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the first round of the nist post-quantum cryptography standardization process. 2019.

[10] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[11] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Krämer, Patrick Longa, and Jefferson E. Ricardini. The lattice-based digital signature scheme qtesla. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 441–460. Springer, 2020.

[12] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020.

[13] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. Frodokem learning with errors key encapsulation (round 3 specifications), 2020. NIST Post-Quantum Cryptography Standardization Project.

[14] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[15] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient Word Size Modular Multiplication over Signed Integers. In *29th IEEE Symposium on Computer Arithmetic, ARITH 2022, Lyon, France, September 12-14, 2022*, pages 94–101. IEEE, 2022.

[16] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[17] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation. 2020.

[18] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting dilithium against leakage: Revisited sensitivity analysis and improved implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):58–79, Aug. 2023.

[19] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[20] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 2018.

[21] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. GALACTICS: gaussian sampling for lattice-based constant- time implementation of cryptographic signatures, revisited, 2019.

[22] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission), 2020. NIST Post-Quantum Cryptography Standardization Project.

[23] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):221–244, 2022.

[24] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 272–293, Cham, 2022. Springer International Publishing. `https://eprint.iacr.org/2022/1243`.

[25] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: practical stateless hash-based signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 368–397. Springer, 2015.

[26] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2129–2146. ACM, 2019.

[27] Daniel J. Bernstein, Tanja Lange, Jonathan Levin, and Bo-Yin Yang. PQ-Connect: Automated post-quantum end-to-end tunnels. Cryptology ePrint Archive, Paper 2024/2092, 2024.

[28] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. XKCP: eXtended Keccak Code Package. `https://github.com/XKCP/XKCP`. commit 7fa59c0.

[29] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. `https://keccak.team/files/Keccak-implementation-3.2.pdf`, 2012. Accessed: 2023-05-26.

[30] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. TurboSHAKE. IACR Cryptol. ePrint Arch., Paper 2023/342, 2023. `https://eprint.iacr.org/2023/342`.

[31] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[32] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019.

[33] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. Post-quantum TLS on embedded systems: Integrating and evaluating kyber and SPHINCS+ with mbed TLS. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 841–852. ACM, 2020.

[34] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M Schanck, Tsunekazu Saito, Peter Schwabe, William Whyte, Keita Xagawa, et al. NTRU: Algorithm specifications and supporting documentation. 2020.

[35] Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter YA Ryan, and Peter B Rønne. Batching csidh group actions using avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 618–649, 2021.

[36] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[37] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[38] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.

[39] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[40] Wouter de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. PhD thesis, Eindhoven University of Technology Eindhoven, The Netherlands, 2015.

[41] Jintai Ding and Albrecht Petzoldt. Current state of multivariate cryptography. *IEEE Secur. Priv.*, 15(4):28–36, 2017.

[42] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, Feb. 2018.

[43] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Ap-*

plication of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV, volume 13794 of *Lecture Notes in Computer Science*, pages 65–94. Springer, 2022.

[44] Arduino Due and ARM Core. Arduino due. *Retrieved*, Jul. 2016.

[45] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015-08-04 2015.

[46] Matthias J. Kannwischer et al. PQM3: Post-quantum crypto library for the ARM Cortex-M3. `https://github.com/mupq/pqm3`.

[47] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93. Springer, 2020.

[48] Apostolos P. Fournaris, Charis Dimopoulos, and Odysseas Koufopavlou. Profiling dilithium digital signature traces for correlation differential side channel attacks. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 281–294, Cham, 2020. Springer International Publishing.

[49] Apostolos P. Fournaris, Charis Dimopoulos, and Odysseas G. Koufopavlou. Profiling dilithium digital signature traces for correlation differential side channel attacks. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation - 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings*, volume 12471 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2020.

[50] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. DPF-ECC: Accelerating elliptic curve cryptography with floating-point computing power of GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–504. IEEE, 2020.

[51] W. Morven Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, volume 29 of *AFIPS Conference Proceedings*, pages 563–578. AFIPS / ACM / Spartan Books, Washington D.C., 1966.

[52] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qtesla. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *Lecture Notes in Computer Science*, pages 74–91. Springer, 2019.

[53] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021.

[54] Bernard Gold and Charles M Rader. Digital processing of signals. *New York: McGraw-Hill*, 1969.

[55] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[56] Denisa Greconici. Kyber on RISC-V. Master's thesis, 2020.

[57] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, 2021.

[58] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):141–164, 2022.

[59] Robert Hackett. IBM plans a huge leap in superfast quantum computing by 2023, 2020.

[60] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. *IEEE Trans. Computers*, 72(4):1055–1066, 2023.

[61] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[62] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting keccak and dilithium implementations on armv7-m. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):1–24, 2024.

[63] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):614–636, 2022.

[64] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. Cryptology ePrint Archive, Paper 2022/956, 2022. `https://eprint.iacr.org/2022/956`.

[65] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *IEEE Transactions on Information Forensics and Security*, 2024.

[66] IBM. IBM quantum breaks the 100-qubit processor barrier — ibm quantum computing blog, November 2021. [Online; accessed 2024-12-21].

[67] IBM. IBM unveils 400 qubit-plus quantum processor and next-generation ibm quantum system two, November 2022. [Online; accessed 2024-12-21].

[68] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 463–481. Springer, 2003.

[69] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[70] Xinyi Ji, Jiankuo Dong, Junhao Huang, Zhijian Yuan, Wangchen Dai, Fu Xiao, and Jingqiang Lin. Eco-crystals: Efficient cryptography crystals on standard risc-v isa. *IEEE Transactions on Computers*, pages 1–13, 2024.

[71] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. IACR Cryptol. ePrint Arch., Paper 2019/844, 2019. `https://eprint.iacr.org/2019/844`.

[72] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, NTRU prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on*

Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021, pages 35–45. IEEE, 2021.

[73] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, ntru prime, and crystals-dilithium. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 35–45, 2021.

[74] Emre Karabulut and Aydin Aysu. FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 691–696. IEEE, 2021.

[75] Çetin Kaya Koç. *About cryptographic engineering*. Springer, 2009.

[76] P Kocher. Differential power analysis. In *Proc. Advances in Cryptology (CRYPTO'99)*, 1999.

[77] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[78] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[79] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[80] Vadim Lyubashevsky. Lattice Signatures without Trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EU-*

ROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.

[81] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 144–155, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[82] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):180–201, 2019.

[83] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium: A small bit-fiddling leak breaks it all. Cryptology ePrint Archive, Paper 2022/106, 2022.

[84] Daniele Micciancio. Cse206a: Lattices algorithms and applications (spring 2014). Lecture notes of a course given in UCSD., 2014. [Online; accessed 2024-12-26].

[85] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, volume 11464 of *Lecture Notes in Computer Science*, pages 344–362. Springer, 2019.

[86] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[87] Peter L Montgomery and Robert D Silverman. An FFT extension to the P-1 factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990.

[88] NIST. Announcing PQC candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, Jul. 2022. (accessed Mar. 13, 2023).

[89] NIST. Post-quantum cryptography: Additional digital signature schemes — csrc, October 2022. [Online; accessed 2024-12-22].

[90] NIST. FIPS 203 (Initial Public Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final, 2024.

[91] NIST. FIPS 204 (Initial Public Draft): Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/final, 2024.

[92] NIST. FIPS 205 (Initial Public Draft): Stateless Hash-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/205/final, 2024.

[93] Chris Peikert. A decade of lattice cryptography. Foundations and trends in theoretical computer science, 2016.

[94] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, page 145–166, Berlin, Heidelberg, 2006. Springer-Verlag.

[95] Rafael del Pino, Thomas Espitau, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. Raccoon: A Side-Channel Secure Signature Scheme. https://raccoonfamily.org/wp-content/uploads/2023/07/raccoon.pdf, 2023. Accessed: 2024-11-23.

[96] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Trans. Emerg. Top. Comput.*, 9(3):1506–1518, 2021.

[97] PQRISCV. PQRISCV: Post-quantum crypto library for the RISC-V. Accessed: Mar. 13, 2023.

[98] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. *Post-Quantum Cryptography Project of NIST*, 2020.

[99] Jiushao Qin. *Mathematical Treatise in Nine Sections*. 1247.

[100] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[101] Azarderakhsh Reza, Campagna Matthew, Costello Craig, and Hess Basil. Supersingular isogeny key encapsulation, 2017. NIST Post-Quantum Cryptography Standardization Project.

[102] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[103] John M. Schanck, Andreas Hulsing, Joost Rijneveld, and Peter Schwabe. NTRU-HRSS-KEM: NIST Round 1 Submission.

[104] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch., Paper 2018/039*, 2018. `http://eprint.iacr.org/2018/039`.

[105] Nicolas Sendrier. Code-based cryptography: State of the art and perspectives. *IEEE Secur. Priv.*, 15(4):44–50, 2017.

[106] SiFive. Sifive FE310-G002 Manual. https://sifive.cdn.prismic.io/sifive/b56b304f-cd2d-421b-9c14-6b35c33f172e_fe310-g002-manual-v1p4.pdf.

[107] François-Xavier Standaert. Introduction to side-channel attacks. *Secure integrated circuits and systems*, pages 27–42, 2010.

[108] Statista. IoT connected devices worldwide 2019-2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, Jul. 2023. (accessed Mar. 13, 2023).

[109] Ko Stoffelen. Efficient cryptography on the RISC-V architecture. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 323–340. Springer, 2019.

[110] Vincent Quentin Ulitzsch, Soundes Marzougui, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium - A small bit-fiddling leak breaks it all. In Benjamin Smith and Huapeng Wu, editors, *Selected Areas in Cryptography - 29th International Conference, SAC 2022, Windsor, ON, Canada, August 24-26, 2022, Revised Selected Papers*, volume 13742 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2022.

[111] Lawrence C Washington. *Elliptic curves: number theory and cryptography.* Chapman and Hall/CRC, 2008.

[112] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, and Yajie et al. Du. Strong quantum computational advantage using a superconducting quantum processor. *Phys. Rev. Lett.*, 127:180501, Oct 2021.

[113] Yanze Yang, Yiran Jia, and Guangwu Xu. On modular algorithms and butterfly operations in number theoretic transform. *Cryptology ePrint Archive*, 2024.

[114] Zewen Ye, Junhao Huang, Tianshun Huang, Yudan Bai, Jinze Li, Hao Zhang, Guangyan Li, Donglong Chen, Ray C. C. Cheung, and Kejie Huang. PQNTRU: acceleration of ntru-based schemes via customized post-quantum processor. *IACR Cryptol. ePrint Arch.*, page 1754, 2024.

[115] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory trade-offs for Saber+ on memory-constrained RISC-V platform. *IEEE Transactions on Computers*, 2022.

[116] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):to appear, 2025.

[117] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 565–595. Springer, 2023.

[118] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. NTRUEncrypt: NIST Round 1 Submission.

LIST OF PUBLICATIONS

1. Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Çetin Kaya Koç, Ray C.C. Cheung, Donglong Chen*, Yet Another Improvement of Plantard Arithmetic for Faster Kyber on Low-End 32-bit IoT Devices[J], *IEEE Transactions on Information Forensics & Security*, 2024, 2024(19): 3800-3813.

2. Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, Donglong Chen*, Revisiting Keccak and Dilithium Implementations on ARMv7-M[J], *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024, 2024(2): 1-24.

3. Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, Donglong Chen*, Improved Plantard Arithmetic for Lattice-based Cryptography[J], *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, 2022(4): 614-636.

4. Junhao Huang, Jipeng Zhang, Weijia Wang, Xuan Yu, Donglong Chen*, Efficient High-order Masking Raccoon on Memory Constrained Devices. [**In Submission**]

5. Zewen Ye, Junhao Huang, Tianshun Huang, Yudan Bai, Jinze Li, Hao Zhang, Guangyan Li, Donglong Chen, Ray CC Cheung, Kejie Huang, PQNTRU: Acceleration of NTRU-based Schemes via Customized Post-Quantum Processor[J], *IEEE Transactions on Computers*, 2025.

6. Haosong Zhao, Junhao Huang, Zihang Chen, Kunxiong Zhu, Donglong Chen, Zhuoran Ji, Hongyuan Liu, VESTA: A Secure and Efficient FHE-based Three-Party Vectorized Evaluation System for Tree Aggregation Models[C], *ACM SIGMETRICS*, 2025.

7. Jipeng Zhang, Yuxing Yan, Junhao Huang, Çetin Kaya Koç*. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}-IM{B}{V}[J]. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025, 2025(1): 632-655.

8. Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, Çetin Kaya Koç*, ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519[C], *Usenix Security*, 2024.

9. Xinyi Ji, Jiankuo Dong, Junhao Huang, Zhijian Yuan, Wangchen Dai, Fu Xiao, Jingqiang Lin, ECO-CRYSTALS: Efficient Cryptography CRYSTALS on Standard RISC-V ISA[J], *IEEE Transactions on Computers*, 2024.

10. Xuan Yu, Jipeng Zhang, Junhao Huang, Donglong Chen, Lu Zhou*, Multi-way High-throughput Implementation of Kyber[C], *Information Security Conference*, 2024: 41-60.

11. Jipeng Zhang, Junhao Huang, Zhe Liu, Sujoy Sinha Roy, Time-memory Trade-offs for Saber+ on Memory-constrained RISC-V, *IEEE Transactions on Computers*, 2022, 2022(71): 2996-3007.

CURRICULUM VITAE

Academic qualifications of the thesis author, Mr. HUANG Junhao:

- Received the degree of Bachelor of Computer Science and Technology from Nanjing University of Aeronautics and Astronautics, June 2018.
- Received the degree of Master of Cyberspace Security from Nanjing University of Aeronautics and Astronautics, April 2021.

March 2025